Hypergolix Python integration documentation

Release 0.1.0

Muterra, Inc

May 19, 2017

Contents

Hypergolix is "programmable Dropbox". Think of it like this:

1. **run** local applications
2. on different computers
3. using **files and folders**
4. synced across the internet

1. **write** local applications
2. on different computers
3. using **programming objects**
4. synced across the internet

Hypergolix runs as a local background service, just like Dropbox does. Once it's running, instead of spending time worrying about relative IP addresses, NAT traversal, pub/sub brokers, or mutual authentication, your code can just do this:

```
>>> import hgx
>>> hgxlink = hgx.HGXLink()
>>> alice = hgxlink.whoami
>>> bob = hgx.Ghid.from_str('AR_2cdgIjlHpaqGa7K8CmvSksaKMIi_scApddFgHT8dZy_
↪vW3YgoUV5T4iVvlzE2V8qsje19K33KZhyI2i0FwAk=')
>>> obj = hgxlink.new_threadsafe(cls=hgx.JsonProxy, state='Hello world!')
>>> obj.share_threadsafe(bob)
```

and Hypergolix takes care of the rest. Alice can modify her object locally, and so long as she and Bob share a common network link (internet, LAN...), Bob will automatically receive an update from upstream.

Hypergolix is marketed towards Internet of Things development, but it's perfectly suitable for other applications as well. For example, the first not-completely-toy Hypergolix demo app is a remote monitoring app for home servers.

CHAPTER 1

Quickstart

This install, configure, and start Hypergolix. You must have already satisfied all install requirements. See *Hypergolix installation* for a thorough install guide, and *Running Hypergolix* for a thorough configuration and startup guide.

Note: There are two parts to Hypergolix: the Hypergolix *daemon*, and the Hypergolix *integration*. The daemon is installed only once per system, but the integration must be installed in every Python environment that wants to use Hypergolix.

Linux & OSX

Installing Hypergolix:

```
sudo apt-get install python3-venv
sudo mkdir /usr/local/hypergolix
sudo python3 -m venv /usr/local/hypergolix/hgx-env
sudo /usr/local/hypergolix/hgx-env/bin/python -m pip install --upgrade pip
sudo /usr/local/hypergolix/hgx-env/bin/pip install hypergolix
sudo ln -s /usr/local/hypergolix/hgx-env/bin/hypergolix /usr/local/bin/hypergolix
hypergolix config --add hgx
hypergolix start app
```

Integration:

```
your/env/here/bin/pip install hgx
```

Windows

Installing Hypergolix:

```
mkdir "%PROGRAMFILES%/Hypergolix"
python -m venv "%PROGRAMFILES%/Hypergolix/hgx-env"
"%PROGRAMFILES%/Hypergolix/hgx-env/Scripts/python" -m pip install --upgrade pip
"%PROGRAMFILES%/Hypergolix/hgx-env/Scripts/pip" install hypergolix
"%PROGRAMFILES%/Hypergolix/hgx-env/Scripts/python" -m hypergolix.winpath ^
↪%PROGRAMFILES^%/Hypergolix/hgx-env/Scripts
set PATH=%PATH%;%PROGRAMFILES%/Hypergolix/hgx-env/Scripts
hypergolix config --add hgx
hypergolix start app
```

Integration:

```
your/env/here/Scripts/pip install hgx
```

CHAPTER 2

Features

Network-agnostic

Both Hypergolix objects and users are hash-addressed. Hypergolix applications don't need to worry about the network topology between endpoints; Hypergolix is offline-first and can failover to local storage and/or LAN servers when internet connectivity is disrupted. This happens completely transparently to both the application and the end user.

Client-side encryption and authentication

All non-local operations are enforced by cryptograpy. Specifically, Hypergolix is backed by the Golix protocol, with the current implementation supporting SHA-512, AES-256, RSA-4096, and X25519, with RSA deprecation planned by early 2018.

Except in local memory, Hypergolix objects are always encrypted (including on-disk). Authentication is verified redundantly (by both client and server), as is integrity. Both checks can be performed offline.

Accounts are self-hosting: all user data is extracted from a special Hypergolix bootstrap object encrypted with the user's `scrypted` password.

Explicit data expiration

Hypergolix data is explicitly removed, and removal propagates to upstream servers automatically. Hypergolix lifetimes are directly analogous to object persistence in a reference-counting memory-managed programming language: each object (Hypergolix container) is referenced by a name (a Hypergolix address), and when all its referents pass out of scope (are explicitly dereferenced), the object itself is garbage collected.

Open source

Hypergolix is completely open-source. Running your own local server is easy: just run the command `hypergolix start server`. Here are some source code links:

- Hypergolix source (~36k LoC)
- Hypergolix event loop management (~4k LoC)
- Hypergolix daemonization (~7k LoC)
- Golix implementation (~5k LoC)

Simple to integrate

Hypergolix supports all major desktop platforms (OSX, Windows, Linux; mobile support is planned in the future). Applications interact with Hypergolix through inter-process communication, using Websockets over `localhost`. Hypergolix ships with Python bindings to the API (broader language support is a very high priority) for easy integration:

```
>>> # This connects to Hypergolix
>>> import hgx
>>> hgxlink = hgx.HGXLink()

>>> # This creates a new object
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.JsonProxy,
...     state = 'Hello World!',
... )

>>> # This updates the object
>>> obj += ' Welcome to Hypergolix.'
>>> obj.hgx_push_threadsafe()
>>> obj
<JsonProxy to 'Hello World! Welcome to Hypergolix!' at Ghid('WFUmW...')>

>>> # This is the object's address, which we need to retrieve it
>>> obj.hgx_ghid.as_str()
'AWFUmWQJvo3U81-hH3WgtXa9bhB9dyXf1QT0yB_l3b6XwjB-WqeN-
↪Lz7JzkMckhDRcjCFS1EmxrcQ1OE2f0Jxh4='

>>> # This retrieves the object later
>>> address = hgx.Ghid.from_str(
...     'AWFUmWQJvo3U81-hH3WgtXa9bhB9dyXf1QT0yB_l3b6XwjB-WqeN-
↪Lz7JzkMckhDRcjCFS1EmxrcQ1OE2f0Jxh4=')
>>> obj = hgxlink.get_threadsafe(cls=hgx.JsonProxy, ghid=address)
>>> obj
<JsonProxy to 'Hello World! Welcome to Hypergolix!' at Ghid('WFUmW...')>
```

CHAPTER 3

Installing and starting Hypergolix

Hypergolix installation

Hypergolix has two parts:

1. the Hypergolix daemon (`pip install hypergolix`)
2. the Hypergolix integration (`pip install hgx`)

To avoid namespace conflicts in dependencies, the daemon should be run in its own dedicated Python environment. One dependency in particular (pycryptodome, used for password `scrypting`) is known to cause issues with shared environments, especially Anaconda. If using Anaconda, be sure to `pip install hypergolix` within a new, **bare** environment.

`hgx`, on the other hand, is a pure Python package, **including its dependencies.** As such, it is much easier to install, and you should almost always use `hgx` in your actual application.

Linux

Additional installation requirements

Hypergolix (via its https://cryptography.io dependency) requires OpenSSL 1.0.2. On Linux, we test against versions 1.0.2j and 1.1.0c. Most recent mainstream Linux distros should ship with a sufficient OpenSSL version, in which case this should be adequate install preparation:

```
sudo apt-get install build-essential libssl-dev libffi-dev python3-dev
```

However, if you get any crypto-related errors, it's likely you need to re-link OpenSSL for `cryptography`, as described here.

For reference, this is our install script for automated testing, which does require some version muckery:

```
if [ -n "${OPENSSL}" ]; then
  OPENSSL_DIR="ossl-1/${OPENSSL}"
  if [[ ! -f "$HOME/$OPENSSL_DIR/bin/openssl" ]]; then
      curl -O https://www.openssl.org/source/openssl-$OPENSSL.tar.gz
      tar zxf openssl-$OPENSSL.tar.gz
      cd openssl-$OPENSSL
      ./config shared no-asm no-ssl2 no-ssl3 -fPIC --prefix="$HOME/$OPENSSL_DIR"
      # modify the shlib version to a unique one to make sure the dynamic
      # linker doesn't load the system one. This isn't required for 1.1.0 at the
      # moment since our Travis builders have a diff shlib version, but it doesn't␣
↪hurt
      sed -i "s/^SHLIB_MAJOR=.*/SHLIB_MAJOR=100/" Makefile
      sed -i "s/^SHLIB_MINOR=.*/SHLIB_MINOR=0.0/" Makefile
      sed -i "s/^SHLIB_VERSION_NUMBER=.*/SHLIB_VERSION_NUMBER=100.0.0/" Makefile
      make depend
      make install

      # Add new openssl to path
      export PATH="$HOME/$OPENSSL_DIR/bin:$PATH"
      export CFLAGS="-I$HOME/$OPENSSL_DIR/include"
      # rpath on linux will cause it to use an absolute path so we don't need to do␣
↪LD_LIBRARY_PATH
      export LDFLAGS="-L$HOME/$OPENSSL_DIR/lib -Wl,-rpath=$HOME/$OPENSSL_DIR/lib"
  fi

  cd $TRAVIS_BUILD_DIR
fi
```

Recommended installation procedure

This will install Hypergolix into a dedicated Python virtual environment within `/usr/local/hypergolix`, and then add the hypergolix command as a symlink in `/usr/local/bin`. Afterwards, Hypergolix should be available directly through the command line by simply typing (for example) `hypergolix start app`.

```
sudo apt-get install python3-venv
sudo mkdir /usr/local/hypergolix
sudo python3 -m venv /usr/local/hypergolix/hgx-env
sudo /usr/local/hypergolix/hgx-env/bin/python -m pip install --upgrade pip
sudo /usr/local/hypergolix/hgx-env/bin/pip install hypergolix
sudo ln -s /usr/local/hypergolix/hgx-env/bin/hypergolix /usr/local/bin/hypergolix
```

Recommended integration procedure

```
path/to/your/app/env/bin/pip install hgx
```

```
#!/path/to/your/app/env/bin/python
import hgx
```

OSX

Additional installation requirements

`Cryptography` ships with compiled binary wheels on OSX, so installation should not require any prerequisites, though it may be necessary to update Python. Additionally, one dependency (`donna25519`) requires the ability to compile C extensions.

Recommended installation procedure

This will install Hypergolix into a dedicated Python virtual environment within `/usr/local/hypergolix`, and then add the hypergolix command as a symlink in `/usr/local/bin`. Afterwards, Hypergolix should be available directly through the command line by simply typing (for example) `hypergolix start app`.

```
sudo apt-get install python3-venv
sudo mkdir /usr/local/hypergolix
sudo python3 -m venv /usr/local/hypergolix/hgx-env
sudo /usr/local/hypergolix/hgx-env/bin/python -m pip install --upgrade pip
sudo /usr/local/hypergolix/hgx-env/bin/pip install hypergolix
sudo ln -s /usr/local/hypergolix/hgx-env/bin/hypergolix /usr/local/bin/hypergolix
```

Recommended integration procedure

```
path/to/your/app/env/bin/pip install hgx
```

```
#!/path/to/your/app/env/bin/python
import hgx
```

Windows

Additional installation requirements

The only Windows prerequisite is Python itself. Because of the namespace conflicts mentioned above, we recommend running Hypergolix in a dedicated virtualenv created through stock Python.

Python is available at Python.org; be sure to download Python 3 (**not** 2.7.xx).

Recommended installation procedure

This will install Hypergolix within your program files. It will then add the Hypergolix bin folder to the *end* of your PATH (meaning everything else will take precedence over it). You will need to run these commands from within an elevated (Administrator) command prompt.

```
mkdir "%PROGRAMFILES%/Hypergolix"
python -m venv "%PROGRAMFILES%/Hypergolix/hgx-env"
"%PROGRAMFILES%/Hypergolix/hgx-env/Scripts/python" -m pip install --upgrade pip
"%PROGRAMFILES%/Hypergolix/hgx-env/Scripts/pip" install hypergolix
"%PROGRAMFILES%/Hypergolix/hgx-env/Scripts/python" -m hypergolix.winpath ^
↪%PROGRAMFILES^%/Hypergolix/hgx-env/Scripts
set PATH=%PATH%;%PROGRAMFILES%/Hypergolix/hgx-env/Scripts
```

Warning: Windows command prompts do not register updates to environment variables after they've been started (they do not handle WS_SettingChange messages). As such, `set PATH=%PATH%;%PROGRAMFILES%/Hypergolix/hgx-env/Scripts` needs to be called in any prompt that was open before Hypergolix installation. Prompts opened afterwards will automatically load the correct `%PATH%`.

Recommended integration procedure

```
path/to/your/app/env/Scripts/pip install hgx
```

```
#!/path/to/your/app/env/Scripts/python
import hgx
```

Building from source

Hypergolix itself is pure Python, so this is easy. Make sure you satisfy the installation requirements listed above, and then clone the source and install it:

```
git clone https://github.com/Muterra/py_hypergolix.git ./hgx-src
/path/to/dest/env/bin/pip install -e ./hgx-src
```

Running Hypergolix

Before running Hypergolix, make sure you have installed it, as described in *Hypergolix installation*.

Configuring Hypergolix

The Hypergolix daemon uses a simple YAML file to store its persistent configuration. This configuration is preferably stored in a subdirectory of your user folder, ie:

- `C:\Users\<username>\.hypergolix\` for Windows systems
- `~/.hypergolix/` for Unix systems

However, Hypergolix can look for the configuration file in other locations as well. Specifically, it searches these locations for `hypergolix.yml`:

1. at the path specified in the environment variable `HYPERGOLIX_HOME`
2. the current directory
3. `~/.hypergolix` (Unix) or `%HOMEPATH%\.hypergolix` (Windows)
4. `/etc/hypergolix` (Unix) or `%LOCALAPPDATA%\Hypergolix` (Windows)

All configuration of Hypergolix is done through this file. When you first run Hypergolix, it will automatically create the following default configuration:

- No remotes (local-only storage)
- Info-level logging
- IPC port 7772

Note: You must restart Hypergolix for any configuration changes to take effect.

Warning: The config file also stores the `user id` and `fingerprint` for the current Hypergolix user. Unless you record your user id somewhere else (for example, in a password manager), if you lose this file, you will probably lose access to your Hypergolix account.

You can also see these values by running the command `hypergolix config --whoami`.

Miscellaneous commands:

```
hypergolix config [-h]
                  [--whoami]
                  [--register]
```

-h, --help	Shows the help message and exits
--whoami	Print the fingerprint and user ID for the current Hypergolix configuration.
--register	Register the current Hypergolix user, allowing them access to the hgx.hypergolix.com remote persister. Requires a web browser.

An example `hypergolix.yml` file:

```
process:
  ghidcache: C:\Users\WinUser\.hypergolix\ghidcache
  logdir: C:\Users\WinUser\.hypergolix\logs
  pid_file: C:\Users\WinUser\.hypergolix\hypergolix.pid
  ipc_port: 7772
instrumentation:
  verbosity: debug
  debug: true
  traceur: false
user:
  fingerprint: AbYly3OIxHORt5knuFOc7rHSj8-
↪x4cihF3Lbf6tabx6WFRUAqV0gGe89dO0pk9ZNOEeDs5XYshcGfv3Z__vxkco=
  user_id: AcTsUOCtK-7iuyLtTnlwL6fgZ7iiw5v2hHmGpHWnoH5pJzECihIDtXn_
↪AoqgrCnYTu0mx4RdAq9ymbzkFPy5zBQ=
  root_secret: null
remotes:
- host: hgx.hypergolix.com
  port: 443
  tls: true
- host: 123.123.123.123
  port: 7770
  tls: false
server:
  ghidcache: C:\Users\WinUser\.hypergolix\ghidcache
  logdir: C:\Users\WinUser\.hypergolix\logs
  pid_file: C:\Users\WinUser\.hypergolix\hgx-server.pid
  host: AUTO
  port: 7770
  verbosity: debug
  debug: true
```

Process configuration:

The Hypergolix process can be customized with specific disk locations. You may also change the localhost port used for inter-process communication with app integrations.

The `ghidcache` directory is used to store the individual Hypergolix objects. It may be the same as the server `ghidcache`.

Warning: Though the Hypergolix app and server may share a `ghidcache` directory, running them from the same directory **at the same time** is currently unsupported, and will thoroughly break Hypergolix.

The `logdir` directory stores a rotating collection of Hypergolix logs. A new log sequence is created every time Hypergolix starts. It may be necessary to periodically empty this directory.

The `pid_file` is used to store the Hypergolix process ID, and to prevent multiple instances of the same Hypergolix process from starting.

The `ipc_port` setting controls which localhost port is used by Hypergolix IPC. It defaults to `7772`.

Warning: Changing the IPC port from the default will require you to always supply the correct `ipc_port` to the *`HGXLink`*.

```
process:
  ghidcache: C:\Users\WinUser\.hypergolix\ghidcache
  logdir: C:\Users\WinUser\.hypergolix\logs
  pid_file: C:\Users\WinUser\.hypergolix\hypergolix.pid
  ipc_port: 7772
```

Instrumentation configuration:

Hypergolix has various instrumentation capabilities to aid in diagnosing problems. All logs are stored locally, in the directory specified in `logdir` above.

Verbosity can be configured between the following values, from quietest to loudest:

1. `error` logs only errors
2. `warning` logs errors and warnings
3. `info` logs errors, warnings, and informational messages
4. `debug` logs all of the above, plus `hypergolix` debug messages
5. `shouty` logs all of the above, plus `websockets` debug messages
6. `extreme` logs all of the above, plus `asyncio` debug messages

Hypergolix can be run in `debug` mode, **which will degrade local performance slightly,** but without it, logged exception tracebacks will be incomplete.

The `traceur` key is currently unused.

```
instrumentation:
  verbosity: debug
  debug: true
  traceur: false
```

User configuration:

The `user` configuration block sets up the Hypergolix user.

Danger: Tampering with this block can render your Hypergolix account unusable!

The `fingerprint` field is your *`Ghid`* fingerprint. Other Hypergolix accounts can use it to share things with you.

The `user_id` field is a *`Ghid`* reference to the object containing your account information, including your private keys. Without it, you cannot access your account.

The `root_secret` field can be used for password-less authentication. We **strongly recommend against using this field** until the login mechanism has been hardened, and even then, it should only be used for semi- or fully-autonomous systems that must survive a system reboot without remote interaction.

```
user:
  fingerprint: AbYly3OIxHORt5knuFOc7rHSj8-
↪x4cihF3Lbf6tabx6WFRUAqV0gGe89dO0pk9ZNOEeDs5XYshcGfv3Z__vxkco=
  user_id: AcTsUOCtK-7iuyLtTnlwL6fgZ7iiw5v2hHmGpHWnoH5pJzECihIDtXn_
↪AoqgrCnYTu0mx4RdAq9ymbzkFPy5zBQ=
  root_secret: null
```

Remote configuration:

Remote persistence servers store Hypergolix data nonlocally. For two Hypergolix accounts to be able to communicate, they must always have at least one persistence server in common.

You can use any combination of remotes you'd like. To use only local storage (*ie*, to use no remotes), set the key to an empty list:

```
remotes: []
```

Otherwise, each remote should be configured as a combination of a host, a port, and a boolean indicator for whether or not the remote server uses TLS:

```
remotes:
- host: hgx.hypergolix.com
  port: 443
  tls: true
```

Server configuration:

The server block allows you to run a remote persistence server on your own machine. It must be started separately (and in addition to) the Hypergolix app.

The `ghidcache` directory is used to store the individual Hypergolix objects. It may be the same as the app `ghidcache`.

Warning: Though the Hypergolix app and server may share a `ghidcache` directory, running them from the same directory **at the same time** is currently unsupported, and will thoroughly break Hypergolix.

The `logdir` directory stores a rotating collection of Hypergolix logs. A new log sequence is created every time Hypergolix starts. It may be necessary to periodically empty this directory.

The `pid_file` is used to store the Hypergolix process ID, and to prevent multiple instances of the same Hypergolix process from starting.

The `host` field determines which hostname the remote server will bind to. By default (including when defined as `null`, it will bind only to `localhost`. If set to `AUTO`, Hypergolix will automatically determine the machine's current IP address, and bind to that. If set to `ANY`, it will bind to any hosts at that port.

The `port` field determines which port the remote server will bind to. It defaults to 7770.

1. `error` logs only errors
2. `warning` logs errors and warnings
3. `info` logs errors, warnings, and informational messages
4. `debug` logs all of the above, plus `hypergolix` debug messages
5. `shouty` logs all of the above, plus `websockets` debug messages
6. `extreme` logs all of the above, plus `asyncio` debug messages

Hypergolix can be run in `debug` mode, **which will degrade local performance slightly,** but without it, logged exception tracebacks will be incomplete.

```
server:
  ghidcache: C:\Users\WinUser\.hypergolix\ghidcache
  logdir: C:\Users\WinUser\.hypergolix\logs
  pid_file: C:\Users\WinUser\.hypergolix\hgx-server.pid
  host: AUTO
  port: 7770
  verbosity: debug
  debug: true
```

Running Hypergolix

Once installed and configured, Hypergolix is easy to use:

```
# Start the app daemon like this
hypergolix start app

# Once started, stop the app daemon like this
hypergolix stop app
```

When you run the Hypergolix app for the first time, it will walk you through the account creation process. After that, Hypergolix will automatically load the existing account, prompting you only for your Hypergolix password.

Warning: If you want Hypergolix to connect with other computers, you must configure remote(s). See above.

Note: Hypergolix is always free to use locally, but on the `hgx.hypergolix.com` remote persistence server, accounts are limited to read-only access (10MB up, unlimited down) until they register. Registration currently costs $10/month.

The Hypergolix server is similarly easy to start. If you want the application daemon to be able to connect to your server on startup, you should start the server first.

```
# Start the server daemon like this
hypergolix start server

# Once started, stop the server daemon like this
hypergolix stop server
```

Note: If you are running a Hypergolix server locally, **please enable logging, with a verbosity of debug,** and consider enabling debug mode. This will help the Hypergolix development team troubleshoot any problems that arise during operation.

Using Hypergolix within your application

As mentioned in *Hypergolix installation*, applications should integrate Hypergolix using the `hgx` package on pip:

```
path/to/your/app/env/bin/pip install hgx
```

From here, develop your application as you normally would, importing hgx and starting the *`HGXLink`*:

```
#!/path/to/your/app/env/bin/python
import hgx
hgxlink = hgx.HGXLink()
```

CHAPTER 4

API reference

Hypergolix addresses: `Ghid`

class **`Ghid`**(*algo*, *address*)

New in version 0.1.

The “Golix hash identifier”: a unique content address for all Golix and Hypergolix content, as defined in the Golix spec. For identities, this is approximately equivalent to their public key fingerprint; for static objects, this is the hash digest of their content; for dynamic objects, this is the hash digest of their dynamic pointers (in Golix terminology, their “bindings”).

Note: `Ghid` instances are hashable and may be used as keys in collections.

Parameters

- **`algo`** (*`int`*) – The Golix-specific integer identifier for the hash algorithm. Currently, only `1` is supported.
- **`address`** (*`bytes`*) – The hash digest of the Ghid.

Raises

- **`ValueError`** – for invalid `algo` s.
- **`ValueError`** – when the length of `address` does not match the expected length for the passed `algo`.

Warning: Once created, changing a `Ghid`‘s `algo` and `address` attributes will break hashing. Avoid doing so. In the future, these attributes will be read-only.

```
>>> from hypergolix import Ghid
>>> ghid = Ghid(1, bytes(64))
>>> ghid
Ghid(algo=1, address=b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
```

algo
The Golix-specific `int` identifier for the hash algorithm.

address
The hash digest of the Ghid, in `bytes`.

Return type bytes

__eq__(*other*)
Compares with another *Ghid* instance.

Parameters **other** (Ghid) – The `Ghid` instance to compare with.

Return type bool

Raises **TypeError** – when attempting to compare with a non-Ghid-like object.

__str__()
Returns a string representation of the `Ghid` *object*, including its class, using a truncated url-safe base64-encoded version of its bytes serialization.

Return type str

```
>>> ghid
Ghid(algo=1, address=b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
>>> str(ghid)
Ghid('AQAAA...')
```

__bytes__()
Serializes the `Ghid` into a Golix-compliant bytestring.

Return type bytes

```
>>> ghid
Ghid(algo=1, address=b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
>>> bytes(ghid)
b
↪'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
```

classmethod from_bytes(*data*)
Loads a `Ghid` from a Golix-compliant bytestring.

Parameters **data** (*bytes*) – The serialization to load

Return type *Ghid*

```
>>> ghid
Ghid(algo=1, address=b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
```

```
>>> bytes(ghid)
b
↪'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
>>> ghid2 = Ghid.from_bytes(b
↪'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
>>> ghid2 == ghid
True
```

as_str()

Returns the raw url-safe base64-encoded version of the Ghid's serialization, without a class identifier.

Return type str

```
>>> ghid
Ghid(algo=1, address=b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
>>> ghid.as_str()

↪'AQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
↪'
```

classmethod from_str(*b64*)

Loads a Ghid from a url-safe base64-encoded Golix-compliant bytestring.

Parameters b64 (*str*) – The serialization to load

Return type *Ghid*

```
>>> ghid
Ghid(algo=1, address=b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
>>> ghid.as_str()

↪'AQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
↪'
>>> ghid3 = Ghid.from_str(
↪'AQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
↪')
>>> ghid3 == ghid
True
```

classmethod pseudorandom(*algo*)

Creates a pseudorandom Ghid for the passed int algorithm identifier.

Parameters b64 (*str*) – The serialization to load

Return type *Ghid*

Warning: This is not suitable for cryptographic purposes. It is primarily useful during testing.

```
>>> ghid
Ghid(algo=1, address=b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
```

```
>>> ghid.as_str()

↪'AQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
↪'
>>> ghid3 = Ghid.from_str(
↪'AQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
↪')
>>> ghid3 == ghid
True
```

Hypergolix IPC: the `HGXLink`

class `HGXLink` (*ipc_port=7772*, *autostart=True*, **args*, *threaded=True*, ***kwargs*)

New in version 0.1.

The inter-process communications link to the Hypergolix service. Uses Websockets over localhost, by default on port 7772. Runs in a dedicated event loop, typically within a separate thread. Must be explicitly stopped during cleanup.

Parameters

- **`ipc_port`** (*`int`*) – The localhost port where the Hypergolix service is currently running.
- **`autostart`** (*`bool`*) – Automatically connect to Hypergolix and start the `HGXLink` during __init__. If `False`, the `HGXLink` must be explicitly started with *`start()`*.
- **`*args`** – Passed to `loopa.TaskCommander`.
- **`threaded`** (*`bool`*) – If `True`, run the `HGXLink` in a separate thread; if `False`, run it in the current thread. In non-threaded mode, the `HGXLink` will block all operations.
- **`**kwargs`** – Passed to `loopa.TaskCommander`.

Returns The `HGXLink` instance.

```
>>> import hgx
>>> hgxlink = hgx.HGXLink()
```

`whoami`

The `Ghid` representing the public key fingerprint of the currently-logged-in Hypergolix user. This address may be used for sharing objects. This attribute is read-only.

Return Ghid if successful

Raises `RuntimeError` – if the Hypergolix service is unavailable.

```
>>> hgxlink.whoami
Ghid(algo=1, address=b'\xf8A\xd6`\x11\xedN\x14\xab\xe5
↪"\x16\x0fs\n\x02\x08\xa1\xca\xa6\xc6
↪$\xa7D\xf7\xb9\xa2\xbc\xc0\x8c\xf3\xe1\xefP\xa1]dE\x87\tw\xb1\xc8\x003\xac>
↪\x89U\xdd\xcc\xb5X\x1d\xcf\x8c\x0e\x0e\x03\x7f\x1e]IQ')
```

`token`

The token for the current application (Python session). Only available after registering the application with the Hypergolix service through one of the *`register_token()`* methods. This attribute is read-only.

Return bytes if the current application has a token.

Raises `RuntimeError` – if the current application has no token.

```
>>> hgxlink.token
AppToken(b'(\x19i\x07&\xff$!h\xa6\x84\xbcr\xd0\xba\xd1')
```

wrap_threadsafe(*callback*)

Wraps a blocking/synchronous function for use as a callback. The wrapped function will be called from within a single-use, dedicated thread from the `HGXLink`'s internal `ThreadPoolExecutor`, so as not to block the `HGXLink` event loop.

This may also be used as a decorator.

```
>>> def threadsafe_callback(obj):
...     print(obj.state)
...
>>> threadsafe_callback
<function threadsafe_callback at 0x00000000051CD620>

>>> # Note that the memory address changes due to wrapping
>>> hgxlink.wrap_threadsafe(threadsafe_callback)
<function threadsafe_callback at 0x00000000051CD6A8>

>>> @hgxlink.wrap_threadsafe
>>> def threadsafe_callback(obj):
...     print(obj.state)
...
>>> threadsafe_callback
<function threadsafe_callback at 0x000000000520B488>
```

wrap_loopsafe(*callback*, *, *target_loop*)

Wraps an asynchronous coroutine for use as a callback. The callback will be run in `target_loop`, which **must be different** from the `HGXLink` event loop (there is no need to wrap callbacks running natively from within the `HGXLink` loop). Use this to have the `HGXLink` run callbacks from within a different event loop (if your application is also using `asyncio` and providing its own event loop).

This may also be used as a decorator.

```
>>> async def loopsafe_callback(obj):
...     print(obj.state)
...
>>> loopsafe_callback
<function loopsafe_callback at 0x0000000005222488>

>>> # Note that the memory address changes due to wrapping
>>> hgxlink.wrap_loopsafe(loopsafe_callback, target_loop=byo_loop)
<function loopsafe_callback at 0x00000000051CD6A8>

>>> @hgxlink.wrap_loopsafe(target_loop=byo_loop)
>>> async def loopsafe_callback(obj):
...     print(obj.state)
...
>>> loopsafe_callback
<function loopsafe_callback at 0x000000000521A228>
```

start()

Starts the HGXLink, connecting to Hypergolix and obtaining the current `whoami`. Must be called explicitly if `autostart` was `False`; otherwise, is called during `HGXLink.__init__`.

```
>>> hgxlink.start()
>>>
```

Note: The following methods each expose three equivalent APIs:

1. an API for the HGXLink event loop, written plainly (ex: *register_token()*).

   Warning: This method **must only** be awaited from within the internal `HGXLink` event loop, or it may break the `HGXLink`, and will likely fail to work.

   This method is a coroutine. Example usage:

   ```
   token = await register_token()
   ```

2. a threadsafe external API, denoted by the _threadsafe suffix (ex: *register_token_threadsafe()*).

   Warning: This method **must not** be called from within the internal `HGXLink` event loop, or it will deadlock.

   This method is a standard, blocking, synchronous method. Example usage:

   ```
   token = register_token_threadsafe()
   ```

3. a loopsafe external API, denoted by the _loopsafe suffix (ex: *register_token_loopsafe()*).

   Warning: This method **must not** be awaited from within the internal `HGXLink` event loop, or it will deadlock.

   This method is a coroutine that may be awaited from your own external event loop. Example usage:

   ```
   token = await register_token_loopsafe()
   ```

stop()
stop_threadsafe()
stop_loopsafe()

Called to stop the `HGXLink` and disconnect from Hypergolix. Must be called before exiting the main thread, or the Python process will not exit, and must be manually halted from an operating system process manager.

new(*cls*, *state*, *api_id=None*, *dynamic=True*, *private=False*)
new_threadsafe(*cls*, *state*, *api_id=None*, *dynamic=True*, *private=False*)
new_loopsafe(*cls*, *state*, *api_id=None*, *dynamic=True*, *private=False*)

Makes a new Hypergolix object.

Parameters

- **cls** (*type*) – the Hypergolix object class to use for this object. See *Basic bytes interface*.
- **state** – the state to initialize the object with. It will be immediately pushed upstream to Hypergolix during creation of the object.
- **api_id** (*bytes*) – the API id to use for this object. If `None`, defaults to the `cls._hgx_DEFAULT_API`.
- **dynamic** (*bool*) – determines whether the created object will be dynamic (and therefore mutable), or static (and wholly immutable).

- **`private`** (*`bool`*) – determines whether the created object will be restricted to **this specific application,** for this specific Hypergolix user. By default, objects created by any Hypergolix application are available to all other Hypergolix apps for the current Hypergolix user.

Returns the created object.

Raises

- **`hypergolix.exceptions.IPCError`** – upon IPC failure, or improper object declaration.
- **`Exception`** – for serialization failures. The specific exception type is determined by the serialization process itself.

```
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.Obj,
...     state = b'Hello world!'
... )
>>> obj
<Obj with state b'Hello world!' at Ghid('Abf3d...')>
>>> # Get the full address to retrieve the object later
>>> obj.ghid.as_str()
'Abf3dRNZAPhrqY93q4Q-wG0QvPnP_
↪anV8XfauVMlFOvAgeC5JVWeXTUftJ6tmYveH0stGaAJ0jN9xKriTT1F6Mk='
```

`get` (*cls*, *ghid*)
`get_threadsafe` (*cls*, *ghid*)
`get_loopsafe` (*cls*, *ghid*)

Retrieves an existing Hypergolix object.

Parameters

- **`cls`** (*`type`*) – the Hypergolix object class to use for this object. See *Basic bytes interface*.
- **`ghid`** (`Ghid`) – the `Ghid` address of the object to retrieve.

Returns the retrieved object.

Raises

- **`hypergolix.exceptions.IPCError`** – upon IPC failure, or improper object declaration.
- **`Exception`** – for serialization failures. The specific exception type is determined by the serialization process itself.

```
>>> address = hgx.Ghid.from_str('Abf3dRNZAPhrqY93q4Q-wG0QvPnP_
↪anV8XfauVMlFOvAgeC5JVWeXTUftJ6tmYveH0stGaAJ0jN9xKriTT1F6Mk=')
>>> obj = hgxlink.get_threadsafe(
...     cls = hgx.ObjBase,
...     ghid = address
... )
>>> obj
<Obj with state b'Hello world!' at Ghid('Abf3d...')>
```

`register_token` (*token=None*)
`register_token_threadsafe` (*token=None*)
`register_token_loopsafe` (*token=None*)

Requests a new application token from the Hypergolix service or re-registers an existing application with the Hypergolix service. If previous instances of the app token have declared a startup object with the Hypergolix service, returns its address.

Tokens can only be registered once per application. Subsequent attempts to register a token will raise `IPCError`. Newly-registered tokens will be available at *`token`*.

App tokens are required for some advanced features of Hypergolix. This token should be reused whenever (and wherever) that exact application is restarted. It is unique for every application, and every Hypergolix user.

Parameters **`token`** (*`hypergolix.utils.AppToken`*) – the application's pre-registered Hypergolix token, or `None` to create one.

Raises **`hypergolix.exceptions.IPCError`** – if unsuccessful.

Return None if no startup object has been declared.

Return hypergolix.Ghid if a startup object has been declared. This is the address of the object, and can be used in a subsequent *`get()`* call to retrieve it.

```
>>> hgxlink.register_token_threadsafe()
>>> hgxlink.token
AppToken(b'(\x19i\x07&\xff$!h\xa6\x84\xbcr\xd0\xba\xd1')

>>> # Some other time, in some other session
>>> app_token = AppToken(b'(\x19i\x07&\xff$!h\xa6\x84\xbcr\xd0\xba\xd1')
>>> hgxlink.register_token_threadsafe(app_token)
```

`register_startup`(*obj*)
`register_startup_threadsafe`(*obj*)
`register_startup_loopsafe`(*obj*)

Registers an object as the startup object. Startup objects are useful to bootstrap configuration, settings, etc. They can be any Hypergolix object, and will be returned to the application at every subsequent call to *`register_token()`*. Startup objects may only be declared after registering an app token.

Parameters **`obj`** – The object to register. May be any Hypergolix object.

Raises **`hypergolix.exceptions.UnknownToken`** – if no token has been registered for the application.

```
>>> obj = hgxlink.new_threadsafe(Obj, state=b'hello world')
>>> hgxlink.register_startup_threadsafe(obj)
```

`deregister_startup`()
`deregister_startup_threadsafe`()
`deregister_startup_loopsafe`()

Registers an object as the startup object. Startup objects are useful to bootstrap configuration, settings, etc. They can be any Hypergolix object, and will be returned to the application at every subsequent call to *`register_token()`*. Startup objects may only be declared after registering an app token.

Raises

- **`hypergolix.exceptions.UnknownToken`** – if no token has been registered for the application.
- **`Exception`** – if no object has be registered for startup.

```
>>> hgxlink.deregister_startup_threadsafe()
```

`register_share_handler`(*api_id*, *handler*)
`register_share_handler_threadsafe`(*api_id*, *handler*)

register_share_handler_loopsafe(*api_id*, *handler*)

Registers a handler for incoming, unsolicited object shares from other Hypergolix users. Without registering a share handler, Hypergolix applications cannot receive shared objects from other users.

The share handler will also be called when other applications from the same Hypergolix user create an object with the appropriate `api_id`.

The share handler callback will be invoked with three arguments: the *Ghid* of the incoming object, the fingerprint *Ghid* of the share origin, and the `hypergolix.utils.ApiID` of the incoming object.

Parameters

- **api_id** (*hypergolix.utils.ApiID*) – determines what objects will be sent to the application. Any objects shared with the current Hypergolix user with a matching api_id will be sent to the application.
- **handler** – the share handler. Unless the `handler` can be used safely from within the `HGXLink` internal event loop, it **must** be wrapped through *wrap_threadsafe()* or *wrap_loopsafe()* prior to registering it as a share handler.

Raises TypeError – If the api_id is not `hypergolix.utils.ApiID` or the handler is not a coroutine (wrap it using *wrap_threadsafe()* or *wrap_loopsafe()* prior to registering it as a share handler).

Warning: Any given API ID can have at most a single share handler per app. Subsequent calls to any of the *register_share_handler()* methods will overwrite the existing share handler without warning.

```
>>> @hgxlink.wrap_threadsafe
... def handler(ghid, origin, api_id):
...     print('Incoming object: ' + str(ghid))
...     print('Sent by: ' + str(origin))
...     print('With API ID: ' + str(api_id))
...
>>> hgxlink.register_share_handler_threadsafe(
...     api_id = hypergolix.utils.ApiID.pseudorandom(),
...     handler = handler
... )
```

Basic `bytes` interface

Note: This assumes familiarity with *Ghid* and *HGXLink* objects.

Hypergolix objects

class Obj(*state*, *api_id*, *dynamic*, *private*, ***, *hgxlink*, *ipc_manager*, *_legroom*, *ghid=None*, *binder=None*, *callback=None*)

New in version 0.1.

The basic Hypergolix object. Create it using *HGXLink.get()* or *HGXLink.new()*; these objects **are not intended to be created directly.** If you create the object directly, it won't receive state updates from upstream.

All Hypergolix objects have a unique, cryptographic-hash-based address (the `Ghid`) and a binder (roughly speaking, the public key fingerprint of the object's creator). They may be dynamic or static.

All Hypergolix objects have a so-called "API ID" – an arbitrary, unique, implicit identifier for the structure of the object. In traditional web service parlance, it's somewhere between an endpoint and a schema, which (unfortunately) is a pretty terrible analogy.

Hypergolix objects persist nonlocally until explicitly deleted through one of the *`delete()`* methods.

Parameters

- **`hgxlink`** (`HGXLink`) – The currently-active *`HGXLink`* object used to connect to the Hypergolix service.
- **`state`** – The state of the object.
- **`api_id`** (*`hgx.utils.ApiID`*) – The API ID for the object (see above).
- **`dynamic`** (*`bool`*) – A value of `True` will result in a dynamic object, whose state may be updated. `False` will result in a static object with immutable state.
- **`private`** (*`bool`*) – Declare the object as available to this application only (as opposed to any application for the logged-in Hypergolix user). Setting this to `True` requires an `HGXLink.app_token`.
- **`ghid`** (`Ghid`) – The `Ghid` address of the object.
- **`binder`** (`Ghid`) – The `Ghid` of the object's binder.

Returns The `Obj` instance, with state declared, **but not initialized with Hypergolix.**

Warning: Hypergolix objects are not intended to be created directly. Instead, they should always be created through the *`HGXLink`*, using one of its *`HGXLink.new()`* or *`HGXLink.get()`* methods.

Creating the objects directly will result in them being unavailable for automatic updates, and forced to poll through their *`sync()`* methods. Furthermore, their *`binder`* and *`ghid`* properties will be unavailable until after the first call to *`push()`*.

```
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.Obj,
...     state = b'Hello world!'
... )
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
```

`state`

The read-write value of the object itself. This will be serialized and uploaded through Hypergolix upon any call to *`push()`*.

Warning: Updating `state` will **not** update Hypergolix. To upload the change, you must explicitly call *`push()`*.

Return type bytes

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.state
```

```
b'Hello world!'
>>> # This change won't yet exist anywhere else
>>> obj.state = b'Hello Hypergolix!'
>>> obj
<Obj with state b'Hello Hypergolix!' at Ghid('bf3dR')>
```

ghid

The read-only address for the object.

Return Ghid read-only address.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.ghid
Ghid(algo=1, address=b'\xb7\xf7u\x13Y\x00\xf8k\xa9\x8fw\xab\x84>
↪\xc0m\x10\xbc\xf9\xcf\xfd\xa9\xd5\xf1w\xda\xb9S%\x14\xeb\xc0\x81\xe0\xb9
↪%U\x9e]5\x1f\xb4\x9e\xad\x99\x8b\xde\x1fK-\x19\xa0\t\xd23}
↪\xc4\xaa\xe2M=E\xe8\xc9')
>>> str(obj.ghid)
Ghid('bf3dR')
```

api_id

The read-only API ID for the object.

Return bytes read-only API ID.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.api_id
ApiID(b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪')
```

private

Whether or not the object is restricted to this application only (see above). Read-only.

Return bool read-only privacy setting.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.private
False
```

dynamic

Is the object dynamic (`True`) or static (`False`)? Read-only.

Return bool read-only dynamic/static status.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.dynamic
True
```

binder

The read-only binder of the object. Roughly speaking, the public key fingerprint of its creator (see above).

Return Ghid read-only binder.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.binder
Ghid(algo=1, address=b'\xf8A\xd6`\x11\xedN\x14\xab\xe5
↪"\x16\x0fs\n\x02\x08\xa1\xca\xa6\xc6
↪$\xa7D\xf7\xb9\xa2\xbc\xc0\x8c\xf3\xe1\xefP\xa1]dE\x87\tw\xb1\xc8\x003\xac>
↪\x89U\xdd\xcc\xb5X\x1d\xcf\x8c\x0e\x0e\x03\x7f\x1e]IQ')
>>> str(obj.binder)
Ghid('fhB1m')
```

callback

Gets, sets, or deletes an update callback. This will be awaited every time the object receives an upstream update, but it will not be called when the application itself calls *push()*. The callback will be passed a single argument: the object itself. The object's *state* will already have been updated to the new upstream state before the callback is invoked.

Because they are running independently of your actual application, and are called by the HGXLink itself, any exceptions raised by the callback will be swallowed and logged.

Parameters **callback** – An awaitable callback.

Warning: For threadsafe or loopsafe usage, this callback must be appropriately wrapped using *HGXLink.wrap_threadsafe()* or *HGXLink.wrap_loopsafe()* **before** setting it as a callback.

Setting the callback:

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> async def handler(obj):
...     print('Updated! ' + repr(obj))
...
>>> obj.callback = handler
```

The resulting call:

```
>>>
Updated! <Obj with state b'Hello Hypergolix!' at Ghid('bf3dR')>
```

__eq__(*other*)

Compares two Hypergolix objects. The result will be True if (and only if) all of the following conditions are satisfied:

1. They both have a *ghid* (else, raise TypeError)
2. The *ghid* compares equally
3. They both have a *state* (else, raise TypeError)
4. The *state* compares equally
5. They both have a *binder* (else, raise TypeError)
6. The *binder* compares equally

Parameters **other** – The Hypergolix object to compare with.

Return bool The comparison result.

Raises **TypeError** – when attempting to compare with a non-Hypergolix object.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj2
<Obj with state b'Hello world!' at Ghid('WFUmW')>
>>> obj == obj2
False
```

Note: The following methods each expose three equivalent APIs:

1.an internal API (ex: *push()*).

> **Warning:** This method **must only** be awaited from within the internal `HGXLink` event loop, or it may break the `HGXLink`, and will likely fail to work.

This method is a coroutine. Example usage:

```
await obj.push()
```

2.a threadsafe API, denoted by the _threadsafe suffix (ex: *push_threadsafe()*).

> **Warning:** This method **must not** be called from within the internal `HGXLink` event loop, or it will deadlock.

This method is a standard, blocking, synchronous method. Example usage:

```
obj.push_threadsafe()
```

3.a loopsafe API, denoted by the _loopsafe suffix (ex: *push_loopsafe()*).

> **Warning:** This method **must not** be awaited from within the internal `HGXLink` event loop, or it will deadlock.

This method is a coroutine that may be awaited from your own external event loop. Example usage:

```
await obj.push_loopsafe()
```

recast(*cls*)
recast_threadsafe(*cls*)
recast_loopsafe(*cls*)

Converts between Hypergolix object types. Returns a new copy of the current Hypergolix object, converted to type `cls`.

Parameters **cls** – the `type` of object to recast into.

Returns a new version of `obj`, in the current class.

> **Warning:** Recasting an object renders the previous Python object inoperable and dead. It will cease to receive updates from the `HGXLink`, and subsequent manipulation of the old object is likely to cause bugs with the new object as well.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.recast_threadsafe(hgx.JsonObj)
<JsonObj with state b'Hello world!' at Ghid('bf3dR')>
```

push()
push_threadsafe()
push_loopsafe()

Notifies the Hypergolix service (through the `HGXLink`) of updates to the object. Must be called explicitly for any changes to be available outside of the current Python session.

Raises

- **hypergolix.exceptions.IPCError** – if unsuccessful.
- **hypergolix.exceptions.LocallyImmutable** – if the object is static, or if the current Hypergolix user did not create it.
- **hypergolix.exceptions.DeadObject** – if the object is unavailable, for example, as a result of a *discard()* call.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> # This state is unknown everywhere except in current memory
>>> obj.state = b'Foo'
>>> obj.state = b'Bar'
>>> # Hypergolix now has no record of b'Foo' ever existing.
>>> obj.push_threadsafe()
>>> # The new state b'Bar' is now known to Hypergolix.
```

sync()
sync_threadsafe()
sync_loopsafe()

Manually initiates an update through Hypergolix. So long as you create and retrieve objects through the `HGXLink`, you will not need these methods.

Raises

- **hypergolix.exceptions.IPCError** – if unsuccessful.
- **hypergolix.exceptions.DeadObject** – if the object is unavailable, for example, as a result of a *discard()* call.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.sync_threadsafe()
```

share(*recipient*)
share_threadsafe(*recipient*)
share_loopsafe(*recipient*)

Shares the `Obj` instance with `recipient`. The recipient will receive a read-only copy of the object, which will automatically update upon any local changes that are *push()*ed upstream.

Parameters **recipient** (Ghid) – The public key fingerprint "identity" of the entity to share with.

Raises

- **hypergolix.exceptions.IPCError** – if immediately unsuccessful.
- **hypergolix.exceptions.DeadObject** – if the object is unavailable, for example, as a result of a *discard()* call.
- **hypergolix.exceptions.Unsharable** – if the object is *private*.

Note: Successful sharing does **not** imply successful receipt. The recipient could ignore the share, be temporarily unavailable, etc.

Note: In order to actually receive the object, the recipient must have a share handler defined for the *api_id* of the object.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> bob = hgx.Ghid.from_str('AfhB1mAR7U4Uq-
↪UiFg9zCgIIocqmxiSnRPe5orzAjPPh71ChXWRFhwl3scgAM6w-iVXdzLVYHc-MDg4Dfx5dSVE=')
>>> obj.share_threadsafe(bob)
```

freeze()
freeze_threadsafe()
freeze_loopsafe()

Creates a static "snapshot" of a dynamic object. This new static object will be available at its own dedicated address.

Returns a frozen copy of the Obj (or subclass) instance. The class of the new instance will match the class of the original.

Raises

- **hypergolix.exceptions.IPCError** – if unsuccessful.
- **hypergolix.exceptions.LocallyImmutable** – if the object is static.
- **hypergolix.exceptions.DeadObject** – if the object is unavailable, for example, as a result of a *discard()* call.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.dynamic
True
>>> frozen = obj.freeze_threadsafe()
>>> frozen
<Obj with state b'hello world' at Ghid('RS48N')>
>>> frozen.dynamic
False
```

hold()
hold_threadsafe()
hold_loopsafe()

Creates a separate binding to the object, preventing its deletion. This does not necessarily prevent other applications at the currently-logged-in Hypergolix user session from removing the object.

Raises

- **hypergolix.exceptions.IPCError** – if unsuccessful.
- **hypergolix.exceptions.DeadObject** – if the object is unavailable, for example, as a result of a *discard()* call.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.hold_threadsafe()
```

discard()
discard_threadsafe()
discard_loopsafe()

Notifies the Hypergolix service that the application is no longer interested in the object, but does not delete it. This renders the object inoperable and dead, preventing most future operations. However, a new copy of the object can still be retrieved through any of the *HGXLink.get()* methods.

Raises

- **hypergolix.exceptions.IPCError** – if unsuccessful.
- **hypergolix.exceptions.DeadObject** – if the object is unavailable, for example, as a result of a *discard()* call.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.discard_threadsafe()
```

delete()
delete_threadsafe()
delete_loopsafe()

Attempts to permanently delete the object. If successful, it will be inoperable and dead. It will also be removed from Hypergolix and made unavailable to other applications, as well as unavailable to any recipients of an *share()* call, unless they have called *hold()*.

Raises

- **hypergolix.exceptions.IPCError** – if unsuccessful.
- **hypergolix.exceptions.DeadObject** – if the object is unavailable, for example, as a result of a *discard()* call.

```
>>> obj
<Obj with state b'Hello world!' at Ghid('bf3dR')>
>>> obj.delete_threadsafe()
```

Hypergolix proxies

class Proxy(*state*, *api_id*, *dynamic*, *private*, ***, *hgxlink*, *ipc_manager*, *_legroom*, *ghid=None*, *binder=None*, *callback=None*)

New in version 0.1.

The Hypergolix proxy, partly inspired by `weakref.proxy`, is a mechanism by which almost any existing Python object can be encapsulated within a Hypergolix-aware wrapper. In every other way, the proxy behaves exactly like the original object. This is accomplished by overloading the `Proxy.__getattr__()`, `Proxy.__setattr__()`, and `Proxy.__delattr__()` methods.

Except where otherwise noted, a Hypergolix `Proxy` exposes the same API as an `Obj`, except that the Hypergolix methods are given an `hgx_` prefix to avoid namespace collisions. For example, `Obj.push()` becomes `Proxy.hgx_push()`, and so on.

A proxy is hashable if its `hgx_ghid` is defined, but unhashable otherwise. Note, however, that this hash has nothing to do with the proxied object. Also note that `isinstance(proxy_obj, collections.Hashable)` will always identify an `Proxy` as hashable, regardless of its actual runtime behavior.

Parameters

- **hgxlink** (`HGXLink`) – The currently-active `HGXLink` object used to connect to the Hypergolix service.
- **state** – The state of the object. For objects using the default (*ie* noop) serialization, this must be `bytes`-like. For subclasses of `Obj`, this can be anything supported by the subclass' serialization strategy.
- **api_id** (*bytes*) – The API ID for the object (see above). Should be a `bytes`-like object of length 64.
- **dynamic** (*bool*) – A value of `True` will result in a dynamic object, whose state may be update. `False` will result in a static object with immutable state.
- **private** (*bool*) – Declare the object as available to this application only (as opposed to any application for the logged-in Hypergolix user). Setting this to `True` requires an `HGXLink.app_token`.
- **ghid** (`Ghid`) – The `Ghid` of the object. Used to instantiate a preexisting object.
- **binder** (`Ghid`) – The `Ghid` of the object's binder. Used to instantiate a preexisting object.

Returns The `Obj` instance, with state declared, **but not initialized with Hypergolix.**

Warning: As with `Obj` objects, `Proxy` objects are not intended to be created directly.

Note: Support for Python special methods (aka "magic methods", "dunder methods", etc) *is* provided. However, due to implementation details in Python itself, this is accomplished by explicitly passing **all** possible `__dunder__` methods *used by Python* to the proxied object.

This has the result that IDEs will present a *very* long list of available methods for `Proxy` objects, even if these methods are not, in fact, available. **However, the built-in `dir()` command should still return a list limited to the methods actually supported by the proxied:proxy combination.**

Note: Proxy objects will detect other `Proxy` instances and subclasses, but **they will not detect `Obj` instances** or subclasses unless they also subclass `Proxy`. This is intentional behavior.

Warning: Because of how Python works, explicitly reassigning `hgx_state` is the only way to reassign the value of the proxied object directly. For example, this will fail, overwriting the name of the object, and leaving the original unchanged:

```
>>> obj
<Proxy to b'Hello world!' at Ghid('bf3dR')>
>>> obj = b'Hello Hypergolix!'
>>> obj
b'Hello Hypergolix!'
```

whereas this will succeed in updating the object state:

```
>>> obj
<Proxy to b'Hello world!' at Ghid('bf3dR')>
>>> obj.hgx_state = b'Hello Hypergolix!'
>>> obj
<Proxy to b'Hello Hypergolix!' at Ghid('bf3dR')>
```

```
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.Proxy,
...     state = b'Hello world!'
... )
>>> obj
<Proxy to b'hello world' at Ghid('bJQMj')>
>>> obj += b' foo'
>>> obj
<Proxy to b'hello world foo' at Ghid('bJQMj')>
>>> obj.state = b'bar'
>>> obj
<Proxy to b'bar' at Ghid('bJQMj')>
```

__eq__(*other*)

Compares the `Proxy` with another object. The comparison recognizes other Hypergolix objects, comparing them more thoroughly than other objects.

If `other` is a Hypergolix object, the comparison will return `True` if and only if:

1. The *`Obj.ghid`* attribute compares equally
2. The *`Obj.state`* attribute compares equally
3. The *`Obj.binder`* attribute compares equally

If, on the other hand, the `other` object is not a Hypergolix object or proxy, it will directly compare `other` with `hgx_state`.

Parameters **`other`** – The object to compare with

Return type bool

```
>>> obj
<Proxy to b'Hello world!' at Ghid('bf3dR')>
>>> obj2
<Proxy to b'Hello world!' at Ghid('WFUmW')>
>>> obj == obj2
False
>>> not_hgx_obj = b'Hello world!'
>>> not_hgx_obj == obj
True
>>> obj2 == not_hgx_obj
True
```

Serialized Python objects

Note: This assumes familiarity with `HGXLink`, `Ghid`, `Obj`, and `Proxy` objects.

JSON serialization

class **`JsonObj`** (*state*, *api_id*, *dynamic*, *private*, ***, *hgxlink*, *ipc_manager*, *_legroom*, *ghid=None*, *binder=None*, *callback=None*)

New in version 0.1.

A Hypergolix object that uses the built-in `json` library for serialization. The resulting string is then encoded in UTF-8. Use it exactly as you would any other `Obj` object.

Warning: `TypeErrors` as a result of improper `state` declarations will not be reported until their value is pushed upstream via `Obj.push()` or equivalent.

```
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.JsonObj,
...     state = 'Hello Json!',
... )
>>> obj
<JsonObj with state 'Hello Json!' at Ghid('bf3dR')>
>>> obj.state = 5
>>> obj
<JsonObj with state 5 at Ghid('bf3dR')>
>>> obj.state = {'seven': 7}
>>> obj
<JsonObj with state {'seven': 7} at Ghid('bf3dR')>
```

class **`JsonProxy`** (*state*, *api_id*, *dynamic*, *private*, ***, *hgxlink*, *ipc_manager*, *_legroom*, *ghid=None*, *binder=None*, *callback=None*)

New in version 0.1.

A Hypergolix proxy that uses the built-in `json` library for serialization. The resulting string is then encoded in UTF-8. Use it exactly as you would any other `Proxy` object.

Warning: `TypeErrors` as a result of improper `hgx_state` declarations will not be reported until their value is pushed upstream via `Obj._hgx_push()` or equivalent.

```
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.JsonProxy,
...     state = 'Hello Json!',
... )
>>> obj
<JsonProxy to 'Hello Json!' at Ghid('bf3dR')>
>>> obj.hgx_state = 5
>>> obj
<JsonProxy to 5 at Ghid('bf3dR')>
>>> obj.hgx_state = {'seven': 7}
>>> obj
<JsonProxy to {'seven': 7} at Ghid('bf3dR')>
```

Pickle serialization

class PickleObj(*state*, *api_id*, *dynamic*, *private*, ***, *hgxlink*, *ipc_manager*, *_legroom*, *ghid=None*, *binder=None*, *callback=None*)

New in version 0.1.

A Hypergolix object that uses the built-in `pickle` library for serialization. The resulting string is then encoded in UTF-8. Use it exactly as you would any other `Obj` object.

Danger: Never use `pickle` to de/serialize objects from an untrusted source. Because `pickle` allows objects to control their own deserialization, retrieving such an object effectively gives the object creator full control over your computer (within the privilege limits of the current Python process).

Warning: `TypeErrors` as a result of improper `state` declarations will not be reported until their value is pushed upstream via `Obj.push()` or equivalent.

```
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.PickleObj,
...     state = 'Hello Pickle!',
... )
>>> obj
<PickleObj with state 'Hello Pickle!' at Ghid('bf3dR')>
>>> obj.state = 5
>>> obj
<PickleObj with state 5 at Ghid('bf3dR')>
>>> obj.state = {'seven': 7}
>>> obj
<PickleObj with state {'seven': 7} at Ghid('bf3dR')>
```

class PickleProxy(*state*, *api_id*, *dynamic*, *private*, ***, *hgxlink*, *ipc_manager*, *_legroom*, *ghid=None*, *binder=None*, *callback=None*)

New in version 0.1.

A Hypergolix proxy that uses the built-in `pickle` library for serialization. The resulting string is then encoded in UTF-8. Use it exactly as you would any other `Proxy` object.

Danger: Never use `pickle` to de/serialize objects from an untrusted source. Because `pickle` allows objects to control their own deserialization, retrieving such an object effectively gives the object creator full control over your computer (within the privilege limits of the current Python process).

Warning: `TypeErrors` as a result of improper `hgx_state` declarations will not be reported until their value is pushed upstream via `Obj.hgx_push()` or equivalent.

```
>>> obj = hgxlink.new_threadsafe(
...     cls = hgx.PickleProxy,
...     state = 'Hello Pickle!',
... )
>>> obj
<PickleProxy to 'Hello Pickle!' at Ghid('bf3dR')>
>>> obj.hgx_state = 5
>>> obj
<PickleProxy to 5 at Ghid('bf3dR')>
>>> obj.hgx_state = {'seven': 7}
>>> obj
<PickleProxy to {'seven': 7} at Ghid('bf3dR')>
```

Custom serialization

Custom serialization of objects can be easily added to Hypergolix by subclassing `Obj` or `Proxy` and overriding:

1. class attribute `_hgx_DEFAULT_API`
2. `staticmethod` or `classmethod` **coroutine** `hgx_pack(state)`
3. `staticmethod` or `classmethod` **coroutine** `hgx_unpack(packed)`

A (non-functional) toy example follows:

```
from hgx.utils import ApiID
from hgx import Obj
from hgx import Proxy

class ToySerializer:
    ''' An Obj that customizes serialization.
    '''
    _hgx_DEFAULT_API = ApiID(bytes(63) + b'\x04')

    @staticmethod
    async def hgx_pack(state):
        ''' Packs the state into bytes.
        '''
        return bytes(state)

    @staticmethod
    async def hgx_unpack(packed):
        ''' Unpacks the state from bytes.
        '''
        return object(packed)


class ToyObj(ToySerializer, Obj):
    pass


class ToyProxy(ToySerializer, Proxy):
    pass
```

CHAPTER 5

Example Hypergolix applications and tutorials

"Telemeter" remote monitoring example application

Telemeter is a very basic Python application. It uses `psutil` to monitor system usage, and then pairs with another Hypergolix identity, broadcasting its system usage on an interval controlled by the remote monitor.

Getting started

After installing Hypergolix, make sure you configure (`hypergolix config --add hgx`) and run it (`hypergolix start app`) on both the monitoring (from now on, the "monitor") and monitored (from now on, the "server") computers. Ideally, also register Hypergolix (`hypergolix config --register`) for the server, to avoid hitting the storage limit for (read-only) free accounts.

Now, set up a project directory and a virtual environment. Don't forget to `pip install hgx` in the environment. I'll use these:

```
mkdir telemeter
cd telemeter
python3 -m venv env
env/bin/pip install hgx
```

Warning: On Windows, replace every `env/bin/pip` or `env/bin/python` with `env/Scripts/pip` and `env/Scripts/python`, respectively.

Hypergolix "Hello world"

To get things started, let's just write a quick application using the blocking (threadsafe) API. It'll just loop forever, recording a timestamp for every loop.

To start, we need to create the Hypergolix inter-process communication link, so we can talk to Hypergolix, and define an interval:

```
class Telemeter:
    ''' Remote monitoring demo app.
    '''

    def __init__(self, interval):
        self.hgxlink = hgx.HGXLink()
        self.interval = interval
```

Now, to set up the app, we want to create a Hypergolix object where we'll store the timestamps. We could make our own serialization system, but Hypergolix ships with JSON objects available, so let's use those:

```
def app_init(self):
    ''' Set up the application.
    '''
    self.status_reporter = self.hgxlink.new_threadsafe(
        cls = hgx.JsonObj,
        state = 'Hello world!'
    )
```

And finally, let's make an app loop to continually update the timestamp until we press `Ctrl+C`:

```
def app_run(self):
    ''' Do the main application loop.
    '''
    while True:
        timestamp = datetime.datetime.now().strftime('%Y.%M.%d @ %H:%M:%S')
        self.status_reporter.state = timestamp
        self.status_reporter.push_threadsafe()
        print('Logged ' + timestamp)
        time.sleep(self.interval)
```

That's all that we need for "Hello world"! Putting it all together, and adding an entry point so we can invoke the script as `env/bin/python telemeter.py`:

```
import time
import datetime
import hgx


class Telemeter:
    ''' Remote monitoring demo app.
    '''

    def __init__(self, interval):
        self.hgxlink = hgx.HGXLink()
        self.interval = interval

        # These are the actual Hypergolix business parts
        self.status_reporter = None

    def app_init(self):
        ''' Set up the application.
        '''
        self.status_reporter = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!'
        )
        print('Created status object: ' + self.status_reporter.ghid.as_str())

    def app_run(self):
        ''' Do the main application loop.
        '''
        while True:
            timestamp = datetime.datetime.now().strftime('%Y.%m.%d @ %H:%M:%S')
            self.status_reporter.state = timestamp
            self.status_reporter.push_threadsafe()
            print('Logged ' + timestamp)
            time.sleep(self.interval)


if __name__ == "__main__":
    try:
        app = Telemeter(interval=5)
        app.app_init()
        app.app_run()

    finally:
        app.hgxlink.stop_threadsafe()
```

Great! Now we have a really simple Hypergolix app. But at the moment, it's not particularly useful – sure, we're logging timestamps, but nobody can see them. Though, if you're feeling particularly adventurous, you could open up a Python interpreter and manually retrieve the status like this:

```
>>> import hgx
>>> hgxlink = hgx.HGXLink()
>>> # Make sure to replace this with the "Created status object: <GHID>"
>>> ghid = hgx.Ghid.from_str('<GHID>')
>>> status_reporter = hgxlink.get_threadsafe(cls=hgx.JsonObj, ghid=ghid)
>>> status_reporter.state
'2016.12.14 @ 09:17:10'
>>> # Wait 5 seconds and...
>>> status_reporter.state
'2016.12.14 @ 09:17:15'
```

If you keep calling `status_reporter.state`, you'll see the timestamp automatically update. Neat! But, we want to do a little more...

A bugfix, plus pairing

We want Telemeter to talk to another computer. To do that, we need to register a share handler. Share handlers tell Hypergolix that an application is available to handle specific kinds of objects from other Hypergolix accounts. But first, if you watched `stdout` closely in the last step, you might have seen a bug:

```
Logged 2016.12.14 @ 09:17:10
Logged 2016.12.14 @ 09:17:15
Logged 2016.12.14 @ 09:17:20
Logged 2016.12.14 @ 09:17:25
Logged 2016.12.14 @ 09:17:31
Logged 2016.12.14 @ 09:17:36
Logged 2016.12.14 @ 09:17:41
Logged 2016.12.14 @ 09:17:46
Logged 2016.12.14 @ 09:17:52
```

Notice how the clock is wandering? The *Obj.push_threadsafe()* takes some time – it needs to talk to the Hypergolix server. A permanent solution might use a generator to constantly generate intervals based on the initial time, but a quick and dirty solution is just to change the `time.sleep` call to compensate for the delay:

```
def app_run(self):
    ''' Do the main application loop.
    '''
    while True:
        timestamp = datetime.datetime.now()
        timestr = timestamp.strftime('%Y.%m.%d @ %H:%M:%S')

        self.status.state = timestr
        self.status.push_threadsafe()

        elapsed = (datetime.datetime.now() - timestamp).total_seconds()
        print('Logged {0} in {1:.3f} seconds.'.format(timestr, elapsed))
        time.sleep(self.interval - elapsed)
```

With that sorted, we can start working on pairing. Thinking a bit about how we want the app to work, we'd like the server to automatically log its status, and for some other computer to occasionally check in on it. But we don't want anyone and everyone to have access to our server's CPU status! So as a quick approximation, let's set up a trust-on-first-connect construct: the first account that connects to the server can watch its status, but any subsequent account cannot.

But first, the server needs to know that the monitor is trying to connect. So we'll define a dedicated pairing object: a small, special object that the monitor can send the server, to request the server's status. To do that, we'll create a specific pairing `API ID`.

Hypergolix uses `API IDs` as a kind of schema identifier for objects. Their meaning is application-specific, but in general you should generate a random API ID using `hgx.utils.ApiID.pseudorandom()` to avoid accidental collisions with other applications. `API IDs` are used in three ways:

1. In general, to explicitly define the object's format and/or purpose
2. For Hypergolix, to dispatch shared objects to applications that have registered share handlers for them
3. For applications, to handle the actual objects

To pair, we're first going to generate (and then, in this case, hard-code) a random `API ID`. We'll use this to identify objects whose sole purpose is for the monitor to announce its existence to the server:

```
PAIR_API = hgx.utils.ApiID(
    b'\x17\n\x12\x17\x03\x0f\x14\x11\x07\x10\x05\x04' +
    b'\x14\x18\x11\x11\x12\x02\x17\x12\x15\x0e\x04' +
    b'\x0f\x11\x19\x07\x19\n\r\x03\x06\x12\x04\x17' +
    b'\x11\x14\x07\t\x08\x13\x19\x04\n\x0f\x15\x12' +
    b'\x14\x07\x19\x16\x13\x18\x0b\x18\x0e\x12\x15\n' +
    b'\n\x16\x0f\x08\x14'
)
```

Now, on the server application, we're going to register a share handler for that `API ID`:

```
def pair_handler(self, ghid, origin, api_id):
    ''' Pair handlers ignore the object itself, instead setting up
    the origin as the paired_fingerprint (unless one already exists,
    in which case it is ignored) and sharing the status object with
    them.

    This also doubles as a way to re-pair the same fingerprint, in
    the event that they have gone offline for a long time and are no
    longer receiving updates.
    '''
    # The initial pairing (pair/trust on first connect)
    if self.paired_fingerprint is None:
        self.paired_fingerprint = origin

    # Subsequent pairing requests from anyone else are ignored
    elif self.paired_fingerprint != origin:
        return

    # Now we want to share the status reporter, if we have one, with the
    # origin
    if self.status_reporter is not None:
        self.status_reporter.share_threadsafe(origin)
```

Share handlers are invoked with the *Ghid* `ghid` of the object being shared, the *Ghid* `origin` of the account that shared it, and the `hgx.utils.ApiID` `api_id` of the object itself. So when our server gets a shared object with the correct `API ID`, it will check to see if it already has a monitor, and, if so, if the pair request is coming from the existing handler (that's the "trust on first connect" bit). If someone else tries to pair, the handler returns immediately, doing nothing. Otherwise, it shares the status object with the monitor.

Now, before we register the share handler (`pair_handler`) with the *HGXLink*, we need to wrap the handler so that the link's internal event loop can `await` it:

```
# Share handlers are called from within the HGXLink event loop, so they
# must be wrapped before use
pair_handler = self.hgxlink.wrap_threadsafe(self.pair_handler)
self.hgxlink.register_share_handler_threadsafe(PAIR_API, pair_handler)
```

Now the server is set up to pair with the monitor, though the monitor can't do anything yet. Putting it all together:

```
import time
import datetime
import hgx


# These are app-specific (here, totally random) API schema identifiers
STATUS_API = hgx.utils.ApiID(
    b'\x02\x0b\x16\x19\x00\x19\x10\x18\x08\x12\x03' +
    b'\x11\x07\x07\r\x0c\n\x14\x04\x13\x07\x04\x06' +
    b'\x13\x01\x0c\x04\x00\x0b\x03\x01\x12\x05\x0f' +
    b'\x01\x0c\x05\x11\x03\x01\x0e\x13\x16\x13\x11' +
    b'\x10\x13\t\x06\x10\x00\x14\x0c\x15\x0b\x07' +
    b'\x0c\x0c\x04\x07\x0b\x0f\x18\x03'
)
PAIR_API = hgx.utils.ApiID(
    b'\x17\n\x12\x17\x03\x0f\x14\x11\x07\x10\x05\x04' +
    b'\x14\x18\x11\x11\x12\x02\x17\x12\x15\x0e\x04' +
    b'\x0f\x11\x19\x07\x19\n\r\x03\x06\x12\x04\x17' +
    b'\x11\x14\x07\t\x08\x13\x19\x04\n\x0f\x15\x12' +
    b'\x14\x07\x19\x16\x13\x18\x0b\x18\x0e\x12\x15\n' +
    b'\n\x16\x0f\x08\x14'
)
```

```
class Telemeter:
    ''' Remote monitoring demo app sender.
    '''

    def __init__(self, interval):
        self.hgxlink = hgx.HGXLink()
        self.interval = interval

        # These are the actual Hypergolix business parts
        self.status = None
        self.paired_fingerprint = None

    def app_init(self):
        ''' Set up the application.
        '''
        self.status = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = STATUS_API
        )

        # Share handlers are called from within the HGXLink event loop, so they
        # must be wrapped before use
        pair_handler = self.hgxlink.wrap_threadsafe(self.pair_handler)
        self.hgxlink.register_share_handler_threadsafe(PAIR_API, pair_handler)

    def app_run(self):
        ''' Do the main application loop.
        '''
        while True:
            timestamp = datetime.datetime.now()
            timestr = timestamp.strftime('%Y.%m.%d @ %H:%M:%S')

            self.status.state = timestr
            self.status.push_threadsafe()

            elapsed = (datetime.datetime.now() - timestamp).total_seconds()
            print('Logged {0} in {1:.3f} seconds.'.format(timestr, elapsed))
            time.sleep(self.interval - elapsed)

    def pair_handler(self, ghid, origin, api_id):
        ''' Pair handlers ignore the object itself, instead setting up
        the origin as the paired_fingerprint (unless one already exists,
        in which case it is ignored) and sharing the status object with
        them.

        This also doubles as a way to re-pair the same fingerprint, in
        the event that they have gone offline for a long time and are no
        longer receiving updates.
        '''
        # The initial pairing (pair/trust on first connect)
        if self.paired_fingerprint is None:
            self.paired_fingerprint = origin

        # Subsequent pairing requests from anyone else are ignored
        elif self.paired_fingerprint != origin:
            return
        # Now we want to share the status reporter, if we have one, with the
        # origin
        if self.status_reporter is not None:
            self.status_reporter.share_threadsafe(origin)


if __name__ == "__main__":
    try:
        app = Telemeter(interval=5)
        app.app_init()
        app.app_run()

    finally:
        app.hgxlink.stop_threadsafe()
```

Pairing, client-side

Status check: the server is ready to broadcast timestamps to the monitor, but the monitor doesn't know how to request, nor receive them. So we'll create a monitor object that creates a pair request on startup:

```
class Monitor:
    ''' Remote monitoring demo app receiver.
    '''

    def __init__(self, telemeter_fingerprint):
        self.hgxlink = hgx.HGXLink()
        self.telemeter_fingerprint = telemeter_fingerprint

        # These are the actual Hypergolix business parts
        self.status = None
        self.pair = None

    def app_init(self):
        ''' Set up the application.
        '''
        # Wait until after registering the share handler to avoid a race
        # condition with the Telemeter
        self.pair = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = PAIR_API
        )
        self.pair.share_threadsafe(self.telemeter_fingerprint)
```

With that, the Monitor can request the server's status. Thus far, our app's logic flow looks like this:

1. Start server telemeter
2. Server logs timestamps, waiting for pairing request
3. Monitor sends pairing request
4. Server responds with the timestamp object

But, the monitor doesn't know how to do anything with the server's timestamp object yet, so let's revisit the monitor. This time around, we'll make use of the *HGXLink*'s native async API for the share handler to make the code a little cleaner:

```
async def status_handler(self, ghid, origin, api_id):
    ''' We sent the pairing, and the Telemeter shared its status obj
    with us in return. Get it, store it locally, and register a
    callback to run every time the object is updated.
    '''
    status = await self.hgxlink.get(
        cls = hgx.JsonObj,
        ghid = ghid
    )
    # This registers the update callback. It will be run in the hgxlink
    # event loop, so if it were blocking/threaded, we would need to wrap
    # it like this: self.hgxlink.wrap_threadsafe(self.update_handler)
    status.callback = self.update_handler
    # We're really only doing this to prevent garbage collection
    self.status = status
```

As before, we need to handle the incoming object's address, origin, and `API ID`. But this time, we want to actually do something with the object: we'll store it locally under `self.status`, and then we register the following simple callback to run every time Hypergolix gets an update for it:

```
async def update_handler(self, obj):
    ''' A very simple, **asynchronous** handler for status updates.
    This will be called every time the Telemeter changes their
    status.
    '''
    print(obj.state)
```

For simplicity's sake, we'll add a busy-wait loop for the monitor, and a small `argparser` to switch between the server "telemeter" and the monitor "telemeter". Don't forget to actually register the share handler (look in `Monitor.app_init`), and then we're good to go! Summing up:

```
import argparse
import time
import datetime
import hgx


# These are app-specific (here, totally random) API schema identifiers
STATUS_API = hgx.utils.ApiID(
    b'\x02\x0b\x16\x19\x00\x19\x10\x18\x08\x12\x03' +
    b'\x11\x07\x07\r\x0c\n\x14\x04\x13\x07\x04\x06' +
    b'\x13\x01\x0c\x04\x00\x0b\x03\x01\x12\x05\x0f' +
    b'\x01\x0c\x05\x11\x03\x01\x0e\x13\x16\x13\x11' +
    b'\x10\x13\t\x06\x10\x00\x14\x0c\x15\x0b\x07' +
    b'\x0c\x0c\x04\x07\x0b\x0f\x18\x03'
)
PAIR_API = hgx.utils.ApiID(
    b'\x17\n\x12\x17\x03\x0f\x14\x11\x07\x10\x05\x04' +
    b'\x14\x18\x11\x11\x12\x02\x17\x12\x15\x0e\x04' +
    b'\x0f\x11\x19\x07\x19\n\r\x03\x06\x12\x04\x17' +
    b'\x11\x14\x07\t\x08\x13\x19\x04\n\x0f\x15\x12' +
    b'\x14\x07\x19\x16\x13\x18\x0b\x18\x0e\x12\x15\n' +
    b'\n\x16\x0f\x08\x14'
)


class Telemeter:
    ''' Remote monitoring demo app sender.
    '''

    def __init__(self, interval):
        self.hgxlink = hgx.HGXLink()
        self.interval = interval

        # These are the actual Hypergolix business parts
        self.status = None
        self.paired_fingerprint = None

    def app_init(self):
        ''' Set up the application.
        '''
        print('My fingerprint is: ' + self.hgxlink.whoami.as_str())
        self.status = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = STATUS_API
        )

        # Share handlers are called from within the HGXLink event loop, so they
        # must be wrapped before use
        pair_handler = self.hgxlink.wrap_threadsafe(self.pair_handler)
        self.hgxlink.register_share_handler_threadsafe(PAIR_API, pair_handler)

    def app_run(self):
        ''' Do the main application loop.
        '''
        while True:
            timestamp = datetime.datetime.now()
            timestr = timestamp.strftime('%Y.%m.%d @ %H:%M:%S')

            self.status.state = timestr
            self.status.push_threadsafe()

            elapsed = (datetime.datetime.now() - timestamp).total_seconds()
            print('Logged {0} in {1:.3f} seconds.'.format(timestr, elapsed))
            time.sleep(self.interval - elapsed)

    def pair_handler(self, ghid, origin, api_id):
        ''' Pair handlers ignore the object itself, instead setting up
        the origin as the paired_fingerprint (unless one already exists,
        in which case it is ignored) and sharing the status object with
        them.

        This also doubles as a way to re-pair the same fingerprint, in
        the event that they have gone offline for a long time and are no
        longer receiving updates.
        '''
        # The initial pairing (pair/trust on first connect)
        if self.paired_fingerprint is None:
            self.paired_fingerprint = origin

        # Subsequent pairing requests from anyone else are ignored
        elif self.paired_fingerprint != origin:
            return

        # Now we want to share the status reporter, if we have one, with the
        # origin
        if self.status is not None:
            self.status.share_threadsafe(origin)


class Monitor:
    ''' Remote monitoring demo app receiver.
    '''

    def __init__(self, telemeter_fingerprint):
        self.hgxlink = hgx.HGXLink()
        self.telemeter_fingerprint = telemeter_fingerprint

        # These are the actual Hypergolix business parts
        self.status = None
        self.pair = None

    def app_init(self):
        ''' Set up the application.
        '''
        # Because we're using a native coroutine for this share handler, it
        # needs no wrapping.
        self.hgxlink.register_share_handler_threadsafe(STATUS_API,
                                                       self.status_handler)

        # Wait until after registering the share handler to avoid a race
        # condition with the Telemeter
        self.pair = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = PAIR_API
        )
        self.pair.share_threadsafe(self.telemeter_fingerprint)

    async def status_handler(self, ghid, origin, api_id):
        ''' We sent the pairing, and the Telemeter shared its status obj
        with us in return. Get it, store it locally, and register a
        callback to run every time the object is updated.
        '''
        status = await self.hgxlink.get(
            cls = hgx.JsonObj,
            ghid = ghid
        )
        # This registers the update callback. It will be run in the hgxlink
        # event loop, so if it were blocking/threaded, we would need to wrap
        # it like this: self.hgxlink.wrap_threadsafe(self.update_handler)
        status.callback = self.update_handler
        # We're really only doing this to prevent garbage collection
        self.status = status

    async def update_handler(self, obj):
        ''' A very simple, **asynchronous** handler for status updates.
        This will be called every time the Telemeter changes their
        status.
        '''
        print(obj.state)

    def app_run(self):
        ''' For now, just busy-wait.
        '''
        while True:
            time.sleep(1)


if __name__ == "__main__":
    argparser = argparse.ArgumentParser(
        description = 'A simple remote telemetry app.'
    )
    argparser.add_argument(
        '--telereader',
        action = 'store',
        default = None,
        help = 'Pass a Telemeter fingerprint to run as a reader.'
    )
    args = argparser.parse_args()

    if args.telereader is not None:
        telemeter_fingerprint = hgx.Ghid.from_str(args.telereader)
        app = Monitor(telemeter_fingerprint)

    else:
        app = Telemeter(interval=5)

    try:
        app.app_init()
        app.app_run()

    finally:
        app.hgxlink.stop_threadsafe()
```

Pairing, client-side

Now that we've got the server and the monitor talking, it's time to make them actually do something worthwhile. First, let's make the logging interval adjustable in the `Telemeter`:

```
INTERVAL_API = hgx.utils.ApiID(
    b'\n\x10\x04\x00\x13\x11\x0b\x11\x06\x02\x19\x00' +
    b'\x11\x12\x10\x10\n\x14\x19\x15\x11\x18\x0f\x0f' +
    b'\x01\r\x0c\x15\x16\x04\x0f\x18\x19\x13\x14\x11' +
    b'\x10\x01\x19\x19\x15\x0b\t\x0e\x15\r\x16\x15' +
    b'\x0e\n\x19\x0b\x14\r\n\x04\x0c\x06\x03\x13\x01' +
    b'\x01\x12\x05'
)

def interval_handler(self, ghid, origin, api_id):
    ''' Interval handlers change our recording interval.
    '''
    # Ignore requests that don't match our pairing.
    # This will also catch un-paired requests.
    if origin != self.paired_fingerprint:
        return

    # If the address matches our pairing, use it to change our interval.
    else:
        # We don't need to create an update callback here, because any
        # upstream modifications will automatically be passed to the
        # object. This is true of all hypergolix objects, but by using a
        # proxy, it mimics the behavior of the int itself.
        interval_proxy = self.hgxlink.get_threadsafe(
            cls = hgx.JsonProxy,
            ghid = ghid
        )
        self._interval = interval_proxy

@property
def interval(self):
    ''' This provides some consumer-side protection against
    malicious interval proxies.
    '''
    try:
        return float(max(self._interval, self.minimum_interval))

    except (ValueError, TypeError):
        return self.minimum_interval
```

And now let's add some code to the `Monitor` to adjust the interval remotely:

```
def set_interval(self, interval):
    ''' Set the recording interval remotely.
    '''
    # This is some supply-side protection of the interval.
    interval = float(interval)

    if self.interval is None:
        self.interval = self.hgxlink.new_threadsafe(
            cls = hgx.JsonProxy,
            state = interval,
            api_id = INTERVAL_API
        )
        self.interval.hgx_share_threadsafe(self.telemeter_fingerprint)
    else:
        # We can't directly reassign the proxy here, because it would just
        # overwrite the self.interval name with the interval float from
        # above. Instead, we need to assign to the state.
        self.interval.hgx_state = interval
        self.interval.hgx_push_threadsafe()
```

Now for a status check. We should be able to run the telemeter and adjust the interval remotely:

```
import argparse
import time
import datetime
import hgx


# These are app-specific (here, totally random) API schema identifiers
STATUS_API = hgx.utils.ApiID(
    b'\x02\x0b\x16\x19\x00\x19\x10\x18\x08\x12\x03' +
    b'\x11\x07\x07\r\x0c\n\x14\x04\x13\x07\x04\x06' +
    b'\x13\x01\x0c\x04\x00\x0b\x03\x01\x12\x05\x0f' +
    b'\x01\x0c\x05\x11\x03\x01\x0e\x13\x16\x13\x11' +
    b'\x10\x13\t\x06\x10\x00\x14\x0c\x15\x0b\x07' +
    b'\x0c\x0c\x04\x07\x0b\x0f\x18\x03'
)
PAIR_API = hgx.utils.ApiID(
    b'\x17\n\x12\x17\x03\x0f\x14\x11\x07\x10\x05\x04' +
    b'\x14\x18\x11\x11\x12\x02\x17\x12\x15\x0e\x04' +
    b'\x0f\x11\x19\x07\x19\n\r\x03\x06\x12\x04\x17' +
    b'\x11\x14\x07\t\x08\x13\x19\x04\n\x0f\x15\x12' +
    b'\x14\x07\x19\x16\x13\x18\x0b\x18\x0e\x12\x15\n' +
    b'\n\x16\x0f\x08\x14'
)
INTERVAL_API = hgx.utils.ApiID(
    b'\n\x10\x04\x00\x13\x11\x0b\x11\x06\x02\x19\x00' +
    b'\x11\x12\x10\x10\n\x14\x19\x15\x11\x18\x0f\x0f' +
    b'\x01\r\x0c\x15\x16\x04\x0f\x18\x19\x13\x14\x11' +
    b'\x10\x01\x19\x19\x15\x0b\t\x0e\x15\r\x16\x15' +
    b'\x0e\n\x19\x0b\x14\r\n\x04\x0c\x06\x03\x13\x01' +
    b'\x01\x12\x05'
)


class Telemeter:
    ''' Remote monitoring demo app sender.
    '''

    def __init__(self, interval, minimum_interval=1):
        self.hgxlink = hgx.HGXLink()
        self._interval = interval
        self.minimum_interval = minimum_interval

        # These are the actual Hypergolix business parts
        self.status = None
        self.paired_fingerprint = None

    def app_init(self):
        ''' Set up the application.
        '''
        print('My fingerprint is: ' + self.hgxlink.whoami.as_str())
        self.status = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = STATUS_API
        )

        # Share handlers are called from within the HGXLink event loop, so they
        # must be wrapped before use
        pair_handler = self.hgxlink.wrap_threadsafe(self.pair_handler)
        self.hgxlink.register_share_handler_threadsafe(PAIR_API, pair_handler)
        # And set up a handler to change our interval
        interval_handler = self.hgxlink.wrap_threadsafe(self.interval_handler)
        self.hgxlink.register_share_handler_threadsafe(INTERVAL_API,
                                                        interval_handler)

    def app_run(self):
        ''' Do the main application loop.
        '''
        while True:
            timestamp = datetime.datetime.now()
            timestr = timestamp.strftime('%Y.%m.%d @ %H:%M:%S')
            self.status.state = timestr
            self.status.push_threadsafe()

            elapsed = (datetime.datetime.now() - timestamp).total_seconds()
            print('Logged {0} in {1:.3f} seconds.'.format(timestr, elapsed))
            # Make sure we clamp this to non-negative values, in case the
            # update took longer than the current interval.
            time.sleep(max(self.interval - elapsed, 0))

    def pair_handler(self, ghid, origin, api_id):
        ''' Pair handlers ignore the object itself, instead setting up
        the origin as the paired_fingerprint (unless one already exists,
        in which case it is ignored) and sharing the status object with
        them.

        This also doubles as a way to re-pair the same fingerprint, in
        the event that they have gone offline for a long time and are no
        longer receiving updates.
        '''
        # The initial pairing (pair/trust on first connect)
        if self.paired_fingerprint is None:
            self.paired_fingerprint = origin

        # Subsequent pairing requests from anyone else are ignored
        elif self.paired_fingerprint != origin:
            return

        # Now we want to share the status reporter, if we have one, with the
        # origin
        if self.status is not None:
            self.status.share_threadsafe(origin)

    def interval_handler(self, ghid, origin, api_id):
        ''' Interval handlers change our recording interval.
        '''
        # Ignore requests that don't match our pairing.
        # This will also catch un-paired requests.
        if origin != self.paired_fingerprint:
            return

        # If the address matches our pairing, use it to change our interval.
        else:
            # We don't need to create an update callback here, because any
            # upstream modifications will automatically be passed to the
            # object. This is true of all hypergolix objects, but by using a
            # proxy, it mimics the behavior of the int itself.
            interval_proxy = self.hgxlink.get_threadsafe(
                cls = hgx.JsonProxy,
                ghid = ghid
            )
            self._interval = interval_proxy

    @property
    def interval(self):
        ''' This provides some consumer-side protection against
        malicious interval proxies.
        '''
        try:
            return float(max(self._interval, self.minimum_interval))

        except (ValueError, TypeError):
            return self.minimum_interval


class Monitor:
    ''' Remote monitoring demo app receiver.
    '''

    def __init__(self, telemeter_fingerprint):
        self.hgxlink = hgx.HGXLink()
        self.telemeter_fingerprint = telemeter_fingerprint

        # These are the actual Hypergolix business parts
        self.status = None
        self.pair = None
        self.interval = None

    def app_init(self):
        ''' Set up the application.
        '''
        # Because we're using a native coroutine for this share handler, it
        # needs no wrapping.
        self.hgxlink.register_share_handler_threadsafe(STATUS_API,
                                                       self.status_handler)

        # Wait until after registering the share handler to avoid a race
        # condition with the Telemeter
        self.pair = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = PAIR_API
        )
        self.pair.share_threadsafe(self.telemeter_fingerprint)

    def app_run(self):
        ''' For now, just busy-wait.
        '''
        while True:
            time.sleep(1)

    async def status_handler(self, ghid, origin, api_id):
        ''' We sent the pairing, and the Telemeter shared its status obj
        with us in return. Get it, store it locally, and register a
        callback to run every time the object is updated.
        '''
        status = await self.hgxlink.get(
            cls = hgx.JsonObj,
            ghid = ghid
        )
        # This registers the update callback. It will be run in the hgxlink
        # event loop, so if it were blocking/threaded, we would need to wrap
        # it like this: self.hgxlink.wrap_threadsafe(self.update_handler)
        status.callback = self.update_handler
        # We're really only doing this to prevent garbage collection
        self.status = status
    async def update_handler(self, obj):
        ''' A very simple, **asynchronous** handler for status updates.
        This will be called every time the Telemeter changes their
        status.
        '''
        print(obj.state)

    def set_interval(self, interval):
        ''' Set the recording interval remotely.
        '''
        # This is some supply-side protection of the interval.
        interval = float(interval)

        if self.interval is None:
            self.interval = self.hgxlink.new_threadsafe(
                cls = hgx.JsonProxy,
                state = interval,
                api_id = INTERVAL_API
            )
            self.interval.hgx_share_threadsafe(self.telemeter_fingerprint)
        else:
            # We can't directly reassign the proxy here, because it would just
            # overwrite the self.interval name with the interval float from
            # above. Instead, we need to assign to the state.
            self.interval.hgx_state = interval
            self.interval.hgx_push_threadsafe()


if __name__ == "__main__":
    argparser = argparse.ArgumentParser(
        description = 'A simple remote telemetry app.'
    )
    argparser.add_argument(
        '--telereader',
        action = 'store',
        default = None,
        help = 'Pass a Telemeter fingerprint to run as a reader.'
    )
    argparser.add_argument(
        '--interval',
        action = 'store',
        default = None,
        type = float,
        help = 'Set the Telemeter recording interval from the Telereader. ' +
               'Ignored by a Telemeter.'
    )
    args = argparser.parse_args()

    if args.telereader is not None:
        telemeter_fingerprint = hgx.Ghid.from_str(args.telereader)
        app = Monitor(telemeter_fingerprint)

        try:
            app.app_init()

            if args.interval is not None:
                app.set_interval(args.interval)
            app.app_run()

        finally:
            app.hgxlink.stop_threadsafe()

    else:
        app = Telemeter(interval=5)

        try:
            app.app_init()
            app.app_run()

        finally:
            app.hgxlink.stop_threadsafe()
```

Enter `psutil`

We've got a simple, adjustable-interval timestamp program running between the monitor and the server. Now let's make the server actually *monitor* something. For this, we'll use `psutil`, a cross-platform system monitoring library.

First we're going to make some quick utilities for formatting purposes. These will make our server logs much easier to read:

```
def humanize_bibytes(n, prefixes=collections.OrderedDict((
                     (0, 'B'),
                     (1024, 'KiB'),
                     (1048576, 'MiB'),
                     (1073741824, 'GiB'),
                     (1099511627776, 'TiB'),
                     (1125899906842624, 'PiB'),
                     (1152921504606846976, 'EiB'),
                     (1180591620717411303424, 'ZiB'),
                     (1208925819614629174706176, 'YiB')))):
    ''' Convert big numbers into easily-human-readable ones.
    '''
    for value, prefix in reversed(prefixes.items()):
        if n >= value:
            return '{:.2f} {}'.format(float(n) / value, prefix)


def format_cpu(cpu_list):
    cpustr = 'CPU:\n----------\n'
    for cpu in cpu_list:
        cpustr += '   ' + str(cpu) + '%\n'
    return cpustr


def format_mem(mem_tup):
    memstr = 'MEM:\n----------\n'
    memstr += '   Avail: ' + humanize_bibytes(mem_tup.available) + '\n'
    memstr += '   Total: ' + humanize_bibytes(mem_tup.total) + '\n'
    memstr += '   Used:  ' + str(mem_tup.percent) + '%\n'
    return memstr
```

```
def format_disk(disk_tup):
    diskstr = 'DISK:\n----------\n'
    diskstr += '  Avail: ' + humanize_bibytes(disk_tup.free) + '\n'
    diskstr += '  Total: ' + humanize_bibytes(disk_tup.total) + '\n'
    diskstr += '  Used:  ' + str(disk_tup.percent) + '%\n'
    return diskstr
```

Great. Now we just need to *slightly* modify the `Telemeter.app_run` method to send our system usage instead of just the timestamp:

```
def app_run(self):
    ''' Do the main application loop.
    '''
    while True:
        timestamp = datetime.datetime.now()
        timestr = timestamp.strftime('%Y.%m.%d @ %H:%M:%S\n==========\n')
        cpustr = format_cpu(psutil.cpu_percent(interval=.1, percpu=True))
        memstr = format_mem(psutil.virtual_memory())
        diskstr = format_disk(psutil.disk_usage('/'))

        status = (timestr + cpustr + memstr + diskstr + '\n')

        self.status.state = status
        self.status.push_threadsafe()

        elapsed = (datetime.datetime.now() - timestamp).total_seconds()
        print('Logged in {:.3f} seconds:\n{}'.format(elapsed, status))
        # Make sure we clamp this to non-negative values, in case the
        # update took longer than the current interval.
        time.sleep(max(self.interval - elapsed, 0))
```

All together now...

```
import argparse
import time
import datetime
import psutil
import collections
import hgx


# These are app-specific (here, totally random) API schema identifiers
STATUS_API = hgx.utils.ApiID(
    b'\x02\x0b\x16\x19\x00\x19\x10\x18\x08\x12\x03' +
    b'\x11\x07\x07\r\x0c\n\x14\x04\x13\x07\x04\x06' +
    b'\x13\x01\x0c\x04\x00\x0b\x03\x01\x12\x05\x0f' +
    b'\x01\x0c\x05\x11\x03\x01\x0e\x13\x16\x13\x11' +
    b'\x10\x13\t\x06\x10\x00\x14\x0c\x15\x0b\x07' +
    b'\x0c\x0c\x04\x07\x0b\x0f\x18\x03'
)
PAIR_API = hgx.utils.ApiID(
    b'\x17\n\x12\x17\x03\x0f\x14\x11\x07\x10\x05\x04' +
    b'\x14\x18\x11\x11\x12\x02\x17\x12\x15\x0e\x04' +
    b'\x0f\x11\x19\x07\x19\n\r\x03\x06\x12\x04\x17' +
    b'\x11\x14\x07\t\x08\x13\x19\x04\n\x0f\x15\x12' +
    b'\x14\x07\x19\x16\x13\x18\x0b\x18\x0e\x12\x15\n' +
    b'\n\x16\x0f\x08\x14'
)
```

```
INTERVAL_API = hgx.utils.ApiID(
    b'\n\x10\x04\x00\x13\x11\x0b\x11\x06\x02\x19\x00' +
    b'\x11\x12\x10\x10\n\x14\x19\x15\x11\x18\x0f\x0f' +
    b'\x01\r\x0c\x15\x16\x04\x0f\x18\x19\x13\x14\x11' +
    b'\x10\x01\x19\x19\x15\x0b\t\x0e\x15\r\x16\x15' +
    b'\x0e\n\x19\x0b\x14\r\n\x04\x0c\x06\x03\x13\x01' +
    b'\x01\x12\x05'
)


def humanize_bibytes(n, prefixes=collections.OrderedDict((
                     (0, 'B'),
                     (1024, 'KiB'),
                     (1048576, 'MiB'),
                     (1073741824, 'GiB'),
                     (1099511627776, 'TiB'),
                     (1125899906842624, 'PiB'),
                     (1152921504606846976, 'EiB'),
                     (1180591620717411303424, 'ZiB'),
                     (1208925819614629174706176, 'YiB')))):
    ''' Convert big numbers into easily-human-readable ones.
    '''
    for value, prefix in reversed(prefixes.items()):
        if n >= value:
            return '{:.2f} {}'.format(float(n) / value, prefix)


def format_cpu(cpu_list):
    cpustr = 'CPU:\n----------\n'
    for cpu in cpu_list:
        cpustr += '   ' + str(cpu) + '%\n'
    return cpustr


def format_mem(mem_tup):
    memstr = 'MEM:\n----------\n'
    memstr += '  Avail: ' + humanize_bibytes(mem_tup.available) + '\n'
    memstr += '  Total: ' + humanize_bibytes(mem_tup.total) + '\n'
    memstr += '  Used:  ' + str(mem_tup.percent) + '%\n'
    return memstr


def format_disk(disk_tup):
    diskstr = 'DISK:\n----------\n'
    diskstr += '  Avail: ' + humanize_bibytes(disk_tup.free) + '\n'
    diskstr += '  Total: ' + humanize_bibytes(disk_tup.total) + '\n'
    diskstr += '  Used:  ' + str(disk_tup.percent) + '%\n'
    return diskstr


class Telemeter:
    ''' Remote monitoring demo app sender.
    '''

    def __init__(self, interval, minimum_interval=1):
        self.hgxlink = hgx.HGXLink()
        self._interval = interval
        self.minimum_interval = minimum_interval
        # These are the actual Hypergolix business parts
        self.status = None
        self.paired_fingerprint = None

    def app_init(self):
        ''' Set up the application.
        '''
        print('My fingerprint is: ' + self.hgxlink.whoami.as_str())
        self.status = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = STATUS_API
        )

        # Share handlers are called from within the HGXLink event loop, so they
        # must be wrapped before use
        pair_handler = self.hgxlink.wrap_threadsafe(self.pair_handler)
        self.hgxlink.register_share_handler_threadsafe(PAIR_API, pair_handler)
        # And set up a handler to change our interval
        interval_handler = self.hgxlink.wrap_threadsafe(self.interval_handler)
        self.hgxlink.register_share_handler_threadsafe(INTERVAL_API,
                                                       interval_handler)

    def app_run(self):
        ''' Do the main application loop.
        '''
        while True:
            timestamp = datetime.datetime.now()
            timestr = timestamp.strftime('%Y.%m.%d @ %H:%M:%S\n==========\n')
            cpustr = format_cpu(psutil.cpu_percent(interval=.1, percpu=True))
            memstr = format_mem(psutil.virtual_memory())
            diskstr = format_disk(psutil.disk_usage('/'))

            status = (timestr + cpustr + memstr + diskstr + '\n')

            self.status.state = status
            self.status.push_threadsafe()

            elapsed = (datetime.datetime.now() - timestamp).total_seconds()
            print('Logged in {:.3f} seconds:\n{}'.format(elapsed, status))
            # Make sure we clamp this to non-negative values, in case the
            # update took longer than the current interval.
            time.sleep(max(self.interval - elapsed, 0))

    def pair_handler(self, ghid, origin, api_id):
        ''' Pair handlers ignore the object itself, instead setting up
        the origin as the paired_fingerprint (unless one already exists,
        in which case it is ignored) and sharing the status object with
        them.

        This also doubles as a way to re-pair the same fingerprint, in
        the event that they have gone offline for a long time and are no
        longer receiving updates.
        '''
        # The initial pairing (pair/trust on first connect)
        if self.paired_fingerprint is None:
            self.paired_fingerprint = origin
        # Subsequent pairing requests from anyone else are ignored
        elif self.paired_fingerprint != origin:
            return

        # Now we want to share the status reporter, if we have one, with the
        # origin
        if self.status is not None:
            self.status.share_threadsafe(origin)

    def interval_handler(self, ghid, origin, api_id):
        ''' Interval handlers change our recording interval.
        '''
        # Ignore requests that don't match our pairing.
        # This will also catch un-paired requests.
        if origin != self.paired_fingerprint:
            return

        # If the address matches our pairing, use it to change our interval.
        else:
            # We don't need to create an update callback here, because any
            # upstream modifications will automatically be passed to the
            # object. This is true of all hypergolix objects, but by using a
            # proxy, it mimics the behavior of the int itself.
            interval_proxy = self.hgxlink.get_threadsafe(
                cls = hgx.JsonProxy,
                ghid = ghid
            )
            self._interval = interval_proxy

    @property
    def interval(self):
        ''' This provides some consumer-side protection against
        malicious interval proxies.
        '''
        try:
            return float(max(self._interval, self.minimum_interval))

        except (ValueError, TypeError):
            return self.minimum_interval


class Monitor:
    ''' Remote monitoring demo app receiver.
    '''

    def __init__(self, telemeter_fingerprint):
        self.hgxlink = hgx.HGXLink()
        self.telemeter_fingerprint = telemeter_fingerprint

        # These are the actual Hypergolix business parts
        self.status = None
        self.pair = None
        self.interval = None

    def app_init(self):
        ''' Set up the application.
        '''
        # Because we're using a native coroutine for this share handler, it
        # needs no wrapping.
        self.hgxlink.register_share_handler_threadsafe(STATUS_API,
                                                       self.status_handler)

        # Wait until after registering the share handler to avoid a race
        # condition with the Telemeter
        self.pair = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = PAIR_API
        )
        self.pair.share_threadsafe(self.telemeter_fingerprint)

    def app_run(self):
        ''' For now, just busy-wait.
        '''
        while True:
            time.sleep(1)

    async def status_handler(self, ghid, origin, api_id):
        ''' We sent the pairing, and the Telemeter shared its status obj
        with us in return. Get it, store it locally, and register a
        callback to run every time the object is updated.
        '''
        status = await self.hgxlink.get(
            cls = hgx.JsonObj,
            ghid = ghid
        )
        # This registers the update callback. It will be run in the hgxlink
        # event loop, so if it were blocking/threaded, we would need to wrap
        # it like this: self.hgxlink.wrap_threadsafe(self.update_handler)
        status.callback = self.update_handler
        # We're really only doing this to prevent garbage collection
        self.status = status

    async def update_handler(self, obj):
        ''' A very simple, **asynchronous** handler for status updates.
        This will be called every time the Telemeter changes their
        status.
        '''
        print(obj.state)

    def set_interval(self, interval):
        ''' Set the recording interval remotely.
        '''
        # This is some supply-side protection of the interval.
        interval = float(interval)

        if self.interval is None:
            self.interval = self.hgxlink.new_threadsafe(
                cls = hgx.JsonProxy,
                state = interval,
                api_id = INTERVAL_API
            )
            self.interval.hgx_share_threadsafe(self.telemeter_fingerprint)
        else:
            # We can't directly reassign the proxy here, because it would just
            # overwrite the self.interval name with the interval float from
            # above. Instead, we need to assign to the state.
            self.interval.hgx_state = interval
            self.interval.hgx_push_threadsafe()


if __name__ == "__main__":
    argparser = argparse.ArgumentParser(
        description = 'A simple remote telemetry app.'
    )
    argparser.add_argument(
        '--telereader',
        action = 'store',
        default = None,
        help = 'Pass a Telemeter fingerprint to run as a reader.'
    )
    argparser.add_argument(
        '--interval',
        action = 'store',
        default = None,
        type = float,
        help = 'Set the Telemeter recording interval from the Telereader. ' +
               'Ignored by a Telemeter.'
    )
    args = argparser.parse_args()

    if args.telereader is not None:
        telemeter_fingerprint = hgx.Ghid.from_str(args.telereader)
        app = Monitor(telemeter_fingerprint)

        try:
            app.app_init()

            if args.interval is not None:
                app.set_interval(args.interval)

            app.app_run()

        finally:
            app.hgxlink.stop_threadsafe()

    else:
        app = Telemeter(interval=5)

        try:
            app.app_init()
            app.app_run()

        finally:
            app.hgxlink.stop_threadsafe()
```

One last thing: daemonizing

Our simple server monitoring app works pretty well, but there's still one problem left: the telemeter cannot run on its own. If we, for example, run it in the background using `python telemeter.py &`, it will shut down as soon as our shell exits (in other words, if we're working via SSH, as soon as our session disconnects). To keep it running, we need to properly daemonize the script.

A full discussion of daemonization is out-of-scope for Hypergolix, but if you want to learn more, check out the Daemoniker documentation. Regardless, the following changes will keep our script running in the background until we explicitly stop it:

```
import argparse
import time
import datetime
import psutil
import collections
import daemoniker
import hgx


# These are app-specific (here, totally random) API schema identifiers
STATUS_API = hgx.utils.ApiID(
    b'\x02\x0b\x16\x19\x00\x19\x10\x18\x08\x12\x03' +
    b'\x11\x07\x07\r\x0c\n\x14\x04\x13\x07\x04\x06' +
    b'\x13\x01\x0c\x04\x00\x0b\x03\x01\x12\x05\x0f' +
    b'\x01\x0c\x05\x11\x03\x01\x0e\x13\x16\x13\x11' +
    b'\x10\x13\t\x06\x10\x00\x14\x0c\x15\x0b\x07' +
    b'\x0c\x0c\x04\x07\x0b\x0f\x18\x03'
)
PAIR_API = hgx.utils.ApiID(
    b'\x17\n\x12\x17\x03\x0f\x14\x11\x07\x10\x05\x04' +
    b'\x14\x18\x11\x11\x12\x02\x17\x12\x15\x0e\x04' +
    b'\x0f\x11\x19\x07\x19\n\r\x03\x06\x12\x04\x17' +
    b'\x11\x14\x07\t\x08\x13\x19\x04\n\x0f\x15\x12' +
    b'\x14\x07\x19\x16\x13\x18\x0b\x18\x0e\x12\x15\n' +
    b'\n\x16\x0f\x08\x14'
)
INTERVAL_API = hgx.utils.ApiID(
    b'\n\x10\x04\x00\x13\x11\x0b\x11\x06\x02\x19\x00' +
    b'\x11\x12\x10\x10\n\x14\x19\x15\x11\x18\x0f\x0f' +
    b'\x01\r\x0c\x15\x16\x04\x0f\x18\x19\x13\x14\x11' +
    b'\x10\x01\x19\x19\x15\x0b\t\x0e\x15\r\x16\x15' +
    b'\x0e\n\x19\x0b\x14\r\n\x04\x0c\x06\x03\x13\x01' +
    b'\x01\x12\x05'
)


def humanize_bibytes(n, prefixes=collections.OrderedDict((
                    (0, 'B'),
                    (1024, 'KiB'),
                    (1048576, 'MiB'),
                    (1073741824, 'GiB'),
                    (1099511627776, 'TiB'),
                    (1125899906842624, 'PiB'),
                    (1152921504606846976, 'EiB'),
                    (1180591620717411303424, 'ZiB'),
                    (1208925819614629174706176, 'YiB')))):
    ''' Convert big numbers into easily-human-readable ones.
    '''
    for value, prefix in reversed(prefixes.items()):
        if n >= value:
            return '{:.2f} {}'.format(float(n) / value, prefix)
```

```
def format_cpu(cpu_list):
    cpustr = 'CPU:\n----------\n'
    for cpu in cpu_list:
        cpustr += '  ' + str(cpu) + '%\n'
    return cpustr


def format_mem(mem_tup):
    memstr = 'MEM:\n----------\n'
    memstr += '  Avail: ' + humanize_bibytes(mem_tup.available) + '\n'
    memstr += '  Total: ' + humanize_bibytes(mem_tup.total) + '\n'
    memstr += '  Used:  ' + str(mem_tup.percent) + '%\n'
    return memstr


def format_disk(disk_tup):
    diskstr = 'DISK:\n----------\n'
    diskstr += '  Avail: ' + humanize_bibytes(disk_tup.free) + '\n'
    diskstr += '  Total: ' + humanize_bibytes(disk_tup.total) + '\n'
    diskstr += '  Used:  ' + str(disk_tup.percent) + '%\n'
    return diskstr


class Telemeter:
    ''' Remote monitoring demo app sender.
    '''

    def __init__(self, interval, minimum_interval=1):
        self.hgxlink = hgx.HGXLink()
        self._interval = interval
        self.minimum_interval = minimum_interval

        # These are the actual Hypergolix business parts
        self.status = None
        self.paired_fingerprint = None

        self.running = True

    def app_init(self):
        ''' Set up the application.
        '''
        # print('My fingerprint is: ' + self.hgxlink.whoami.as_str())
        self.status = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = STATUS_API
        )

        # Share handlers are called from within the HGXLink event loop, so they
        # must be wrapped before use
        pair_handler = self.hgxlink.wrap_threadsafe(self.pair_handler)
        self.hgxlink.register_share_handler_threadsafe(PAIR_API, pair_handler)
        # And set up a handler to change our interval
        interval_handler = self.hgxlink.wrap_threadsafe(self.interval_handler)
        self.hgxlink.register_share_handler_threadsafe(INTERVAL_API,
                                                       interval_handler)

    def app_run(self):
        ''' Do the main application loop.
        '''
        while self.running:
            timestamp = datetime.datetime.now()
            timestr = timestamp.strftime('%Y.%m.%d @ %H:%M:%S\n==========\n')
            cpustr = format_cpu(psutil.cpu_percent(interval=.1, percpu=True))
            memstr = format_mem(psutil.virtual_memory())
            diskstr = format_disk(psutil.disk_usage('/'))

            status = (timestr + cpustr + memstr + diskstr + '\n')

            self.status.state = status
            self.status.push_threadsafe()

            elapsed = (datetime.datetime.now() - timestamp).total_seconds()
            # print('Logged in {:.3f} seconds:\n{}'.format(elapsed, status))
            # Make sure we clamp this to non-negative values, in case the
            # update took longer than the current interval.
            time.sleep(max(self.interval - elapsed, 0))

    def signal_handler(self, signum):
        self.running = False
        self.hgxlink.stop_threadsafe()

    def pair_handler(self, ghid, origin, api_id):
        ''' Pair handlers ignore the object itself, instead setting up
        the origin as the paired_fingerprint (unless one already exists,
        in which case it is ignored) and sharing the status object with
        them.

        This also doubles as a way to re-pair the same fingerprint, in
        the event that they have gone offline for a long time and are no
        longer receiving updates.
        '''
        # The initial pairing (pair/trust on first connect)
        if self.paired_fingerprint is None:
            self.paired_fingerprint = origin

        # Subsequent pairing requests from anyone else are ignored
        elif self.paired_fingerprint != origin:
            return

        # Now we want to share the status reporter, if we have one, with the
        # origin
        if self.status is not None:
            self.status.share_threadsafe(origin)

    def interval_handler(self, ghid, origin, api_id):
        ''' Interval handlers change our recording interval.
        '''
        # Ignore requests that don't match our pairing.
        # This will also catch un-paired requests.
        if origin != self.paired_fingerprint:
            return

        # If the address matches our pairing, use it to change our interval.
        else:
            # We don't need to create an update callback here, because any
            # upstream modifications will automatically be passed to the
            # object. This is true of all hypergolix objects, but by using a
            # proxy, it mimics the behavior of the int itself.
            interval_proxy = self.hgxlink.get_threadsafe(
                cls = hgx.JsonProxy,
                ghid = ghid
            )
            self._interval = interval_proxy

    @property
    def interval(self):
        ''' This provides some consumer-side protection against
        malicious interval proxies.
        '''
        try:
            return float(max(self._interval, self.minimum_interval))

        except (ValueError, TypeError):
            return self.minimum_interval


class Monitor:
    ''' Remote monitoring demo app receiver.
    '''

    def __init__(self, telemeter_fingerprint):
        self.hgxlink = hgx.HGXLink()
        self.telemeter_fingerprint = telemeter_fingerprint

        # These are the actual Hypergolix business parts
        self.status = None
        self.pair = None
        self.interval = None

    def app_init(self):
        ''' Set up the application.
        '''
        # Because we're using a native coroutine for this share handler, it
        # needs no wrapping.
        self.hgxlink.register_share_handler_threadsafe(STATUS_API,
                                                       self.status_handler)

        # Wait until after registering the share handler to avoid a race
        # condition with the Telemeter
        self.pair = self.hgxlink.new_threadsafe(
            cls = hgx.JsonObj,
            state = 'Hello world!',
            api_id = PAIR_API
        )
        self.pair.share_threadsafe(self.telemeter_fingerprint)

    def app_run(self):
        ''' For now, just busy-wait.
        '''
        while True:
            time.sleep(1)

    async def status_handler(self, ghid, origin, api_id):
        ''' We sent the pairing, and the Telemeter shared its status obj
        with us in return. Get it, store it locally, and register a
        callback to run every time the object is updated.
        '''
        print('Incoming status: ' + ghid.as_str())
        status = await self.hgxlink.get(
            cls = hgx.JsonObj,
            ghid = ghid
        )
        # This registers the update callback. It will be run in the hgxlink
        # event loop, so if it were blocking/threaded, we would need to wrap
        # it like this: self.hgxlink.wrap_threadsafe(self.update_handler)
        status.callback = self.update_handler
        # We're really only doing this to prevent garbage collection
        self.status = status

    async def update_handler(self, obj):
        ''' A very simple, **asynchronous** handler for status updates.
        This will be called every time the Telemeter changes their
        status.
        '''
        print(obj.state)

    def set_interval(self, interval):
        ''' Set the recording interval remotely.
        '''
        # This is some supply-side protection of the interval.
        interval = float(interval)

        if self.interval is None:
            self.interval = self.hgxlink.new_threadsafe(
                cls = hgx.JsonProxy,
                state = interval,
                api_id = INTERVAL_API
            )
            self.interval.hgx_share_threadsafe(self.telemeter_fingerprint)
        else:
            # We can't directly reassign the proxy here, because it would just
            # overwrite the self.interval name with the interval float from
            # above. Instead, we need to assign to the state.
            self.interval.hgx_state = interval
            self.interval.hgx_push_threadsafe()


if __name__ == "__main__":
    argparser = argparse.ArgumentParser(
        description = 'A simple remote telemetry app.'
    )
    argparser.add_argument(
        '--telereader',
        action = 'store',
        default = None,
        help = 'Pass a Telemeter fingerprint to run as a reader.'
    )
    argparser.add_argument(
        '--interval',
        action = 'store',
        default = None,
        type = float,
        help = 'Set the Telemeter recording interval from the Telereader. ' +
               'Ignored by a Telemeter.'
    )
    argparser.add_argument(
        '--pidfile',
        action = 'store',
        default = 'telemeter.pid',
        type = str,
        help = 'Set the name for the PID file for the Telemeter daemon.'
    )
    argparser.add_argument(
        '--stop',
        action = 'store_true',
        help = 'Stop an existing Telemeter daemon.'
    )
    args = argparser.parse_args()

    # This is the READER
    if args.telereader is not None:
        telemeter_fingerprint = hgx.Ghid.from_str(args.telereader)
        app = Monitor(telemeter_fingerprint)

        try:
            app.app_init()

            if args.interval is not None:
                app.set_interval(args.interval)

            app.app_run()

        finally:
            app.hgxlink.stop_threadsafe()

    # This is the SENDER, but we're stopping it.
    elif args.stop:
        daemoniker.send(args.pidfile, daemoniker.SIGTERM)

    # This is the SENDER, and we're starting it.
    else:
        # We need to actually daemonize the app so that it persists without
        # an SSH connection
        with daemoniker.Daemonizer() as (is_setup, daemonizer):
            is_parent, pidfile = daemonizer(
                args.pidfile,
                args.pidfile,
                strip_cmd_args = False
            )

            # Parent exits here

        # Just the child from here
        app = Telemeter(interval=5)

        try:
            sighandler = daemoniker.SignalHandler1(
                pidfile,
                sigint = app.signal_handler,
                sigterm = app.signal_handler,
                sigabrt = app.signal_handler
            )
            sighandler.start()

            app.app_init()
            app.app_run()

        finally:
            app.hgxlink.stop_threadsafe()
```

Index