# HTCondor Python Bindings

## Problem Statement

The most common way to interact with HTCondor is via the command-line tools. This method of interaction is problematic in higher-level languages such as Python. Issues include:

1. **Wrappers re-implemented many times poorly**. There is no standard way to interact with HTCondor via Python; hence, most projects write their own wrappers around the CLI. Each one has at least one bug and none have achieved complete coverage of the client tools.
2. **High barrier of entry**. ClassAds, as a complete language with complex quoting and evaluation rules, takes a large mount of code to re-implement. If a sysadmin wants to read in a ClassAd, make a few modifications, and write out a corresponding ClassAd, they must choose between ignoring large parts of the language or writing hundreds of lines of code. See <https://htcondor-wiki.cs.wisc.edu/index.cgi/tktview?tn=3380> for more detail.
3. **Lack of ClassAds**. The ClassAd language has several powerful concepts that are not modeled correctly in Python. For the most part, python wrappers limit themselves to literals in the ClassAd language (removing most of the utility of ClassAds) and cast the ClassAd to a dictionary.
4. **Poor error management**. Errors and exceptional cases are modeled poorly in the CLI. There are, for example, 6 defined reasons that a Collector query can fail; these are enumerated in the code, but not in the CLI. At best, python wrappers currently can communicate “successful” versus “not successful”. At worst, they try to scrape the error message from stdout – this “screen scraping” is error prone, fragile, and limits HTCondor’s ability to make improvements.
5. **Poor efficiency**. Security sessions cannot be reused between CLI invocations. Many OS resources are wasted (c.f. the glideinWMS issue in having to close 50k file descriptors before exec, for example; each CLI invocation re-loads the configuration subsystem, which may be very expensive) when forking many CLI tools.

## Proposed Architecture

I propose adding python bindings as a part of the core HTCondor distribution. These would be distributed by default in the UW and the PROPER builds.

I propose exporting a set of client classes oriented toward the daemons they interact with. For the initial version, two classes are exposed:

* **Collector**: Allows one to locate daemons, do generic ad queries, and
* **Schedd**: Allows one to query for jobs, act on jobs (remove, hold, release, suspend, continue), submit jobs, and edit jobs.

As desired, interaction with other daemons (master, negotiator) may be added. We aim to expose direct interfaces with minimal amounts of client-side logic. For example, the *Schedd.submit* method will take a ClassAd object. It will not aim to replicate the eight thousand lines of logic contained in condor\_submit for parsing and transforming a submit file.

The bindings will be built using boost.python, which allows C++ interfaces to be directly invoked by python. This approach improves on top of other approaches (SWIG, native Python C API) as it allows us to use high-level C++ concepts (throwing exceptions, objects, polymorphism, python types in C++) that map cleanly into python. This saves implementation and maintenance time, as well as providing a more natural python developer experience.

Interfaces with memory ownership issues will not be exported to python. This may means some ClassAd semantics will not be available or be slightly changed. For example, lists and parent scopes in ClassAds cannot be utilized safely.

The python bindings may limit the exported interfaces to provide better safety to the user. For example, the qmgmt transaction and connection interface will not be exposed directly to the user as it can block or crash the upstream schedd if used incorrectly. It will only be accessible indirectly via the Schedd.edit / Schedd.submit method implementations. This will limit the user to one edit or one submitted cluster per transaction, but will reduce the ability for inadvertent abuse.

The python bindings will not be thread-safe.

These bindings will add a build-time dependency on the python development libraries (which are available on all supported platforms) and a run-time dependency on python and the boost python libraries when the bindings are used. The UW build can ship and link a private boost python if desired.

## Implementation

The initial implementation will be from the python-condor and python-classad modules (<https://github.com/bbockelm/python-condor> and <https://github.com/bbockelm/python-classad>). It is about one thousand lines of C++ code; it will need to be imported and integrated with the Condor build process.

All of the initial client classes described in the prior section are already implemented.

Currently, no Linux-specific mechanisms are used in the code. It may be possible to support Windows out-of-the-box; if it does not immediately work, compilation on that platform will be disabled.

Unit tests will be done using the python unittest library; they should be straightforward if it is easy to provide a personal Condor scaffold for the server-side.

## Development Plan

* Integrate with Condor build [1 day].
* Debug or disable failed builds on non-RHEL platforms [1 day].
* Write unit tests [3 days].