

XRootD client plug-ins

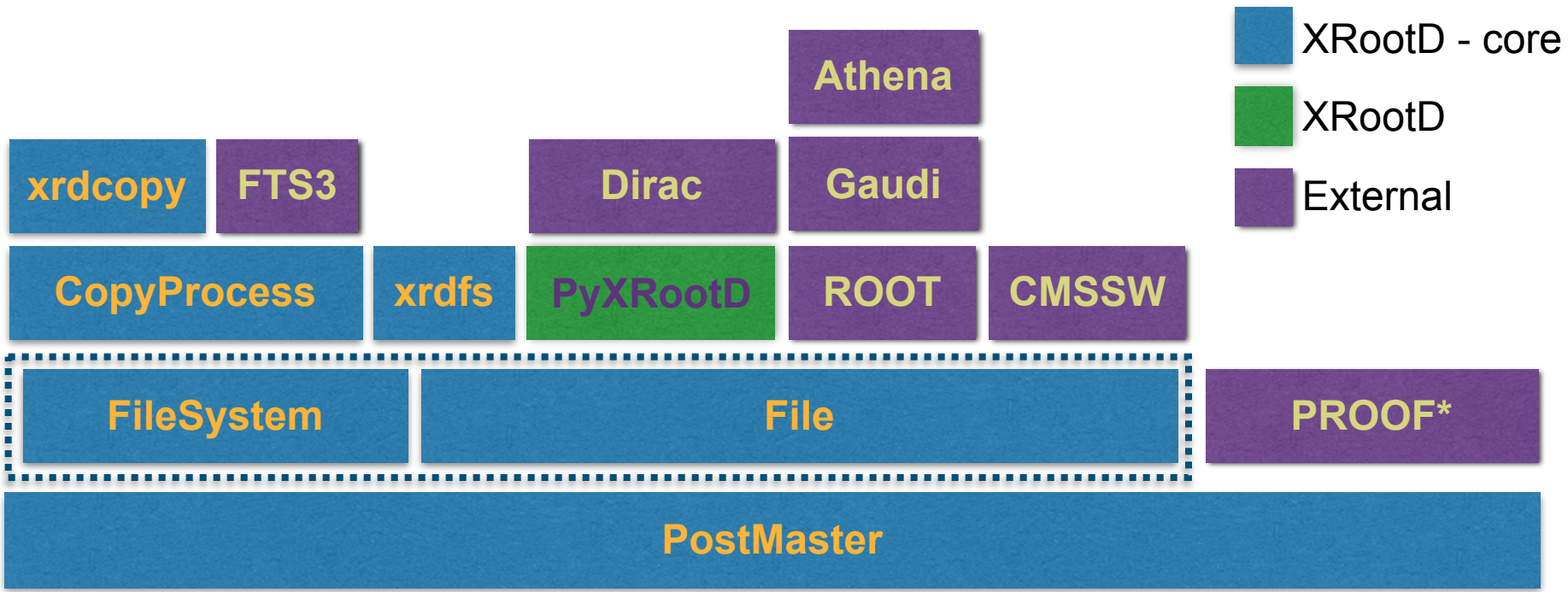
Łukasz Janyst

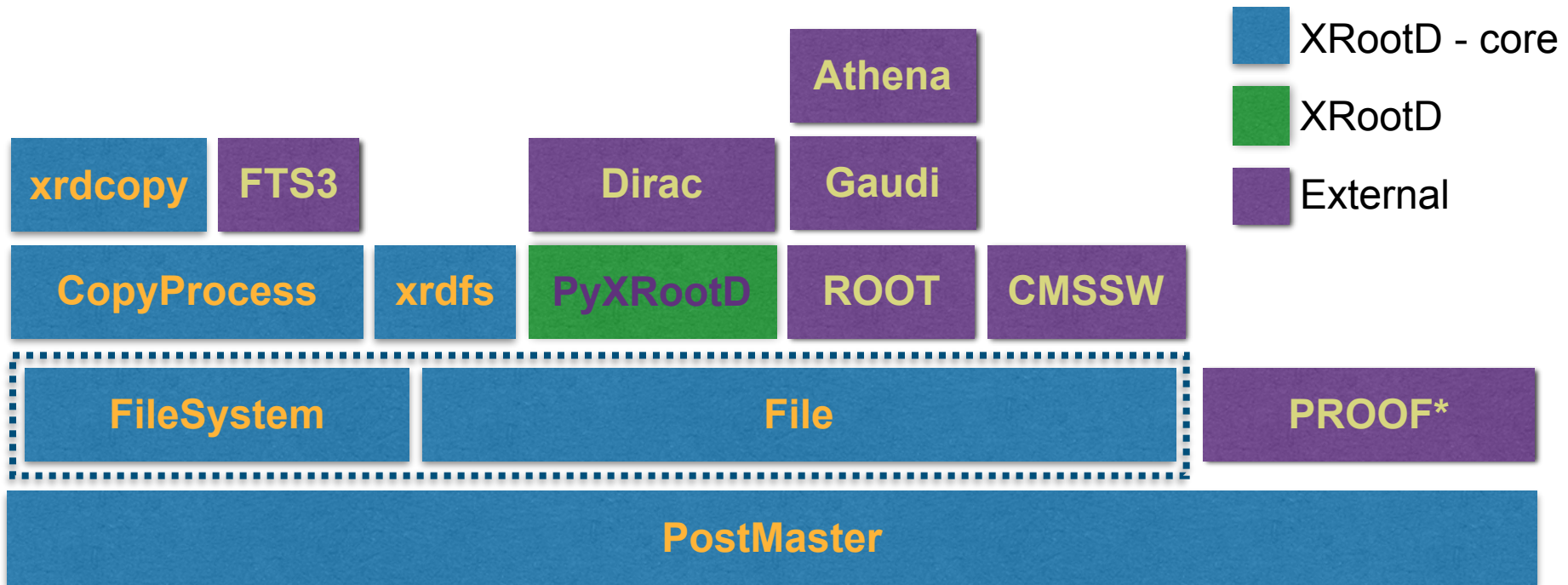


- XrdCI and its API stack
- The plug-in mechanism:
 - Where?
 - What?
 - How?
 - Why?/What for?
 - When?

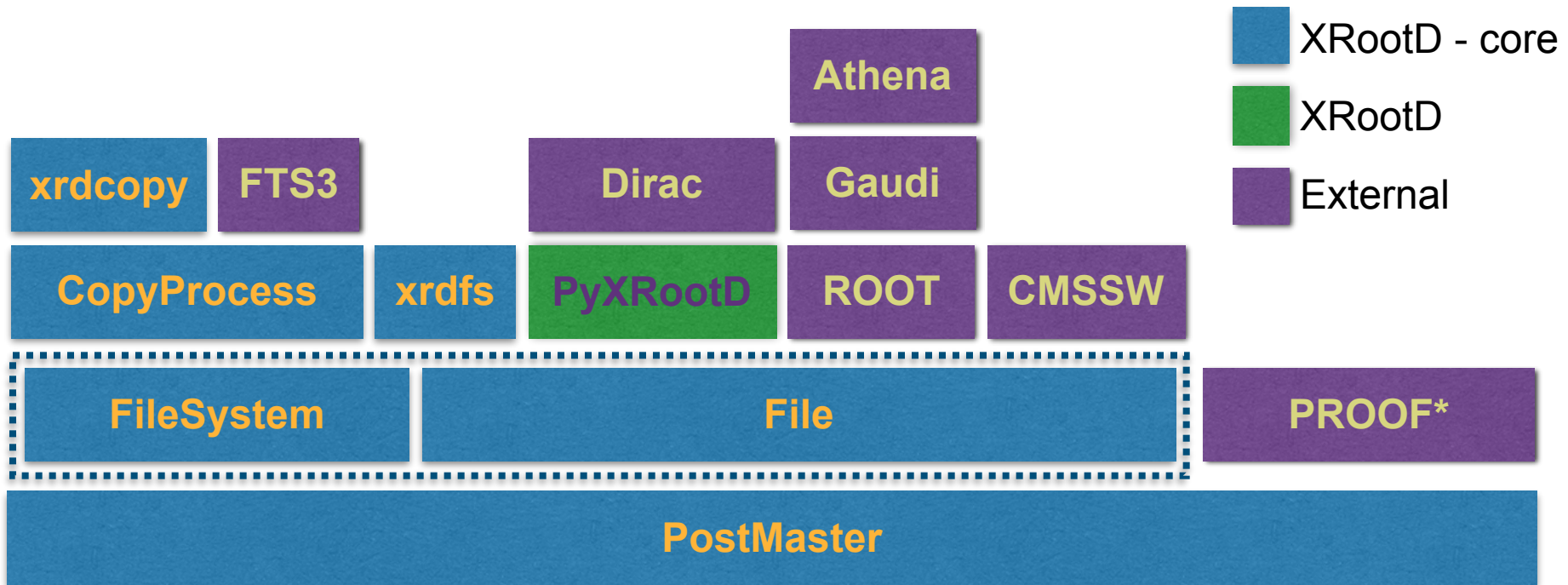


XrdCI API stack & neighbours

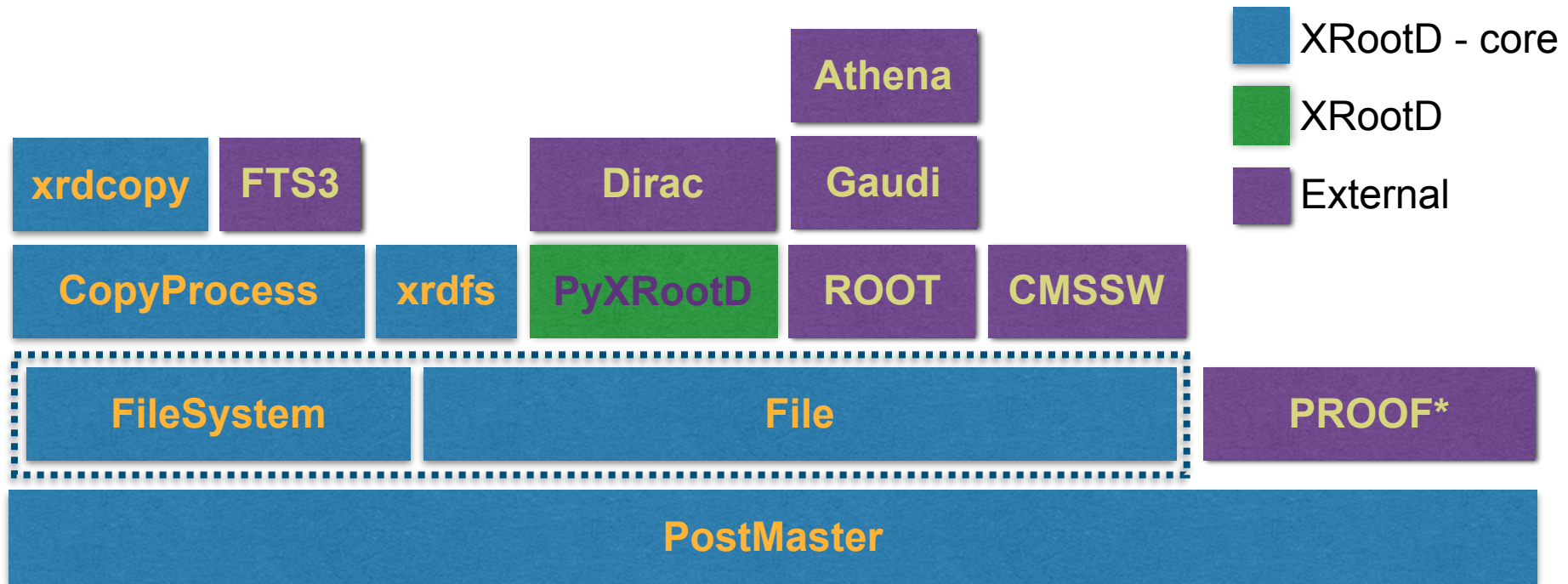




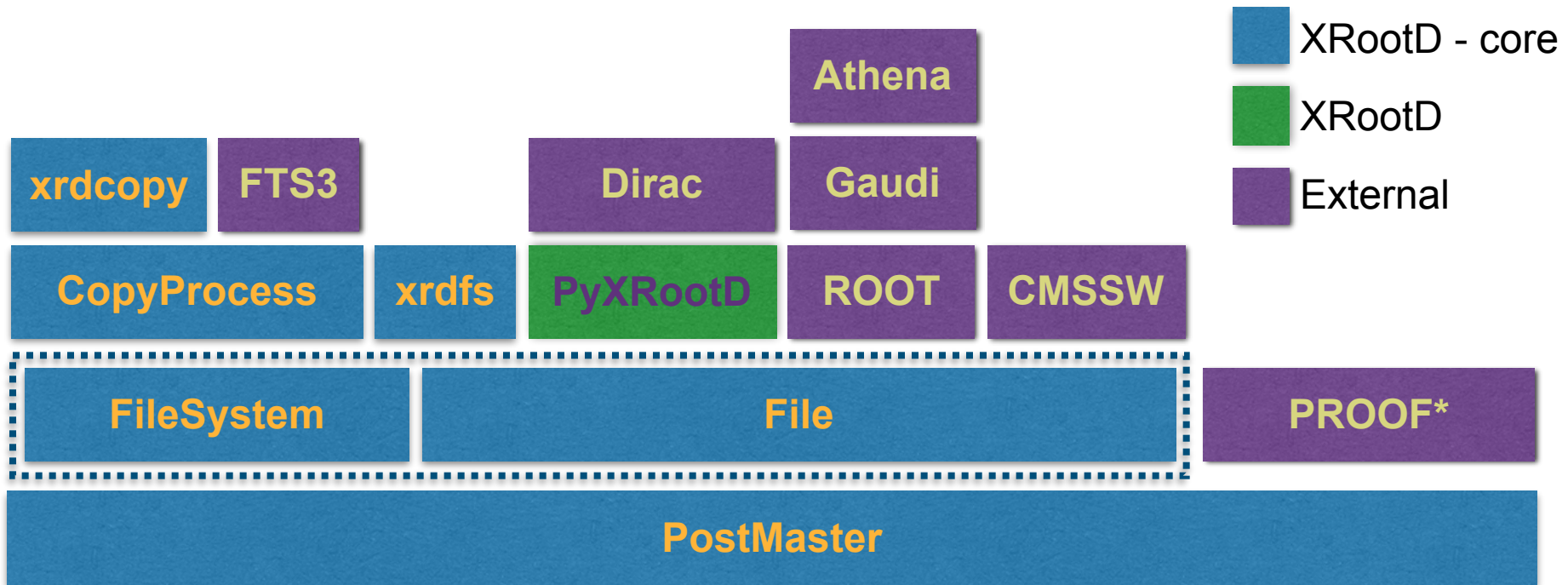
- **PostMaster** - low-level message handling API
 - sends messages
 - asynchronous - notifies message handlers about sent/incoming messages
 - notifies about stream status changes (disconnections)



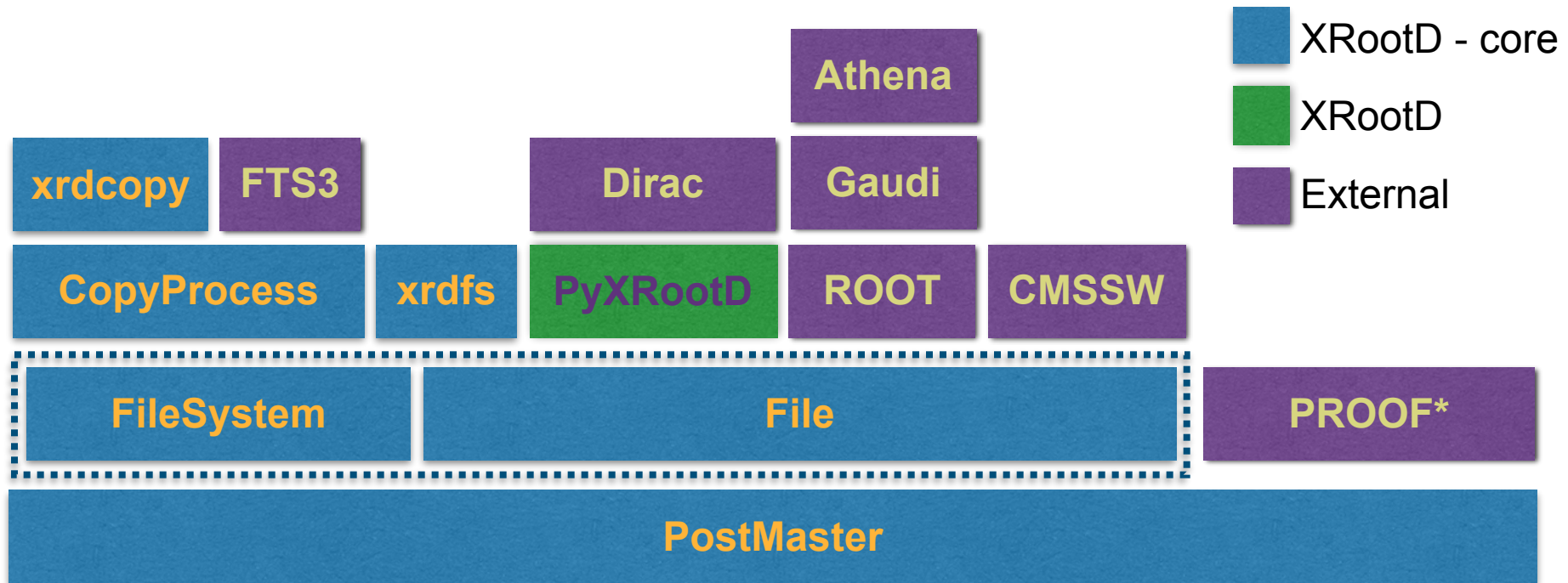
- **File/FileSystem** - implement XRootD operations
 - user-facing C++ API
 - does reads, writes, mkdirs, listings, staging and the like
 - asynchronous - call back when response is read



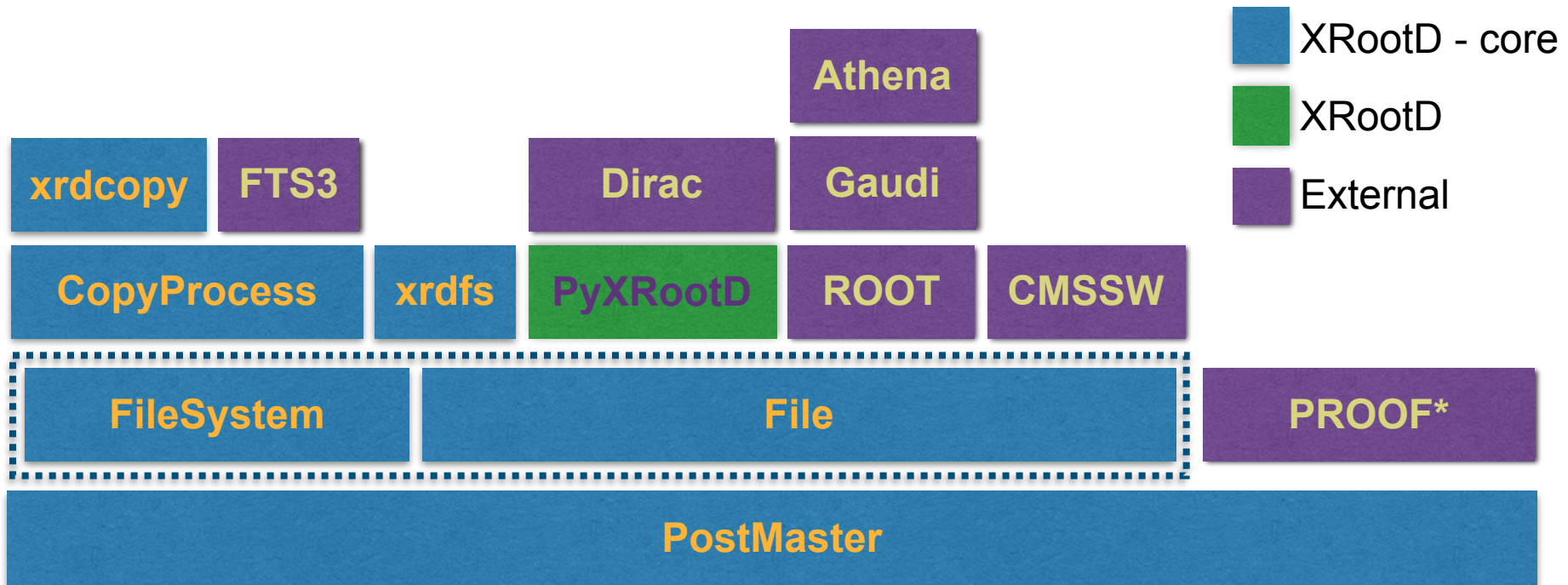
- **CopyProcess** - implement copy operations
 - user-facing C++ API, a library call
 - take source/target URLs, couple more parameters and do the magic
 - notify periodically about progress using a call-back



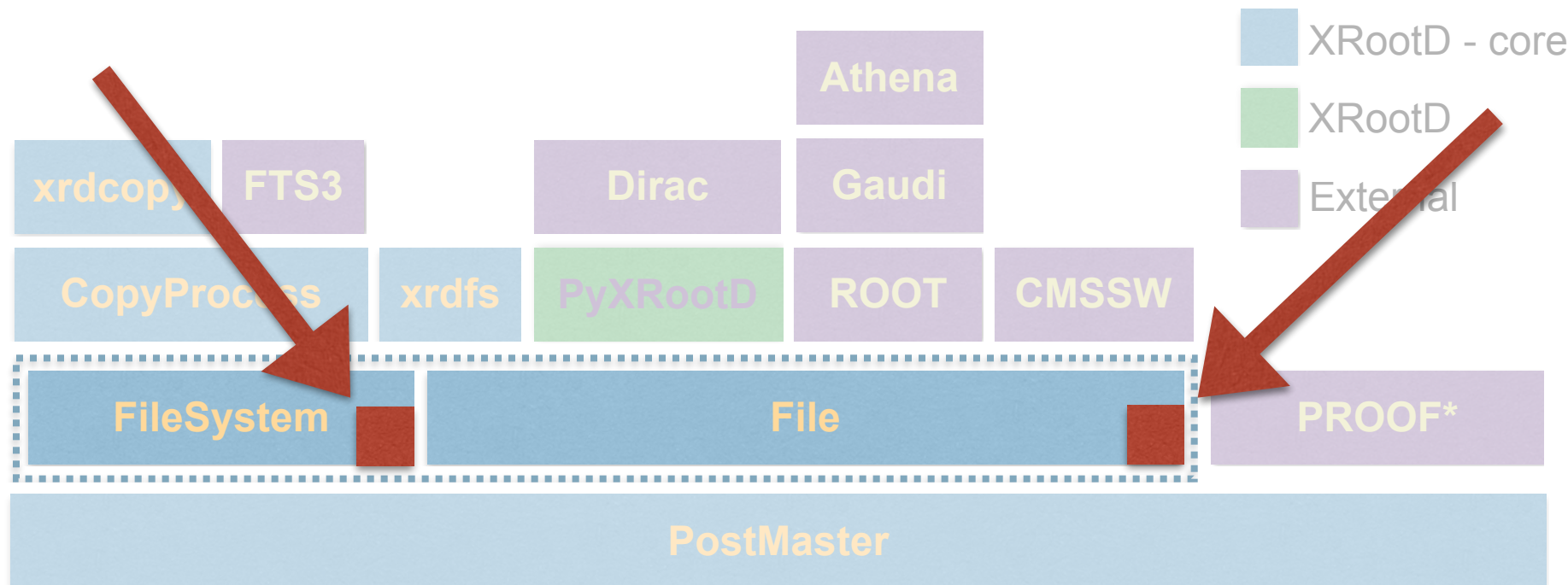
- **xrdcopy** - the copy command
 - translate command line parameters to CopyProcess calls
- **xrdfs** - the meta-data command
 - translate command line parameters to FileSystem calls



- **PyXRootD** - make the user APIs available in Python
 - all APIs user APIs available: File, FileSystem and CopyProcess
 - uses Python callables to handle call-backs



- **Neighbours** - all the purple boxes above and quite a bit more that did not fit the diagram
 - **EOS** uses a bit of everything
 - significant number of packages use **XrdPosix** interface



- **Plug-ins** - replace the original implementation
 - both for File and FileSystem objects
 - all calls can be replaced
 - all the layers above can benefit without changing a single line of code

- All xroot protocol requests are implemented as asynchronous methods
- The calls queue the request and return, **never block**

```
XRootDStatus File::Open( const std::string &url,  
                        OpenFlags::Flags   flags,  
                        Access::Mode       mode,  
                        ResponseHandler    *handler,  
                        uint16_t           timeout )
```

- The response handler is called when the response is ready
- Synchronous versions implemented in terms of asynchronous ones, with a semaphore

- The plug-in API is exactly the same - except for the **virtual** keyword
- Only asynchronous calls may be overloaded

```
virtual
```

```
XRootDStatus File::Open( const std::string &url,  
                          OpenFlags::Flags   flags,  
                          Access::Mode      mode,  
                          ResponseHandler    *handler,  
                          uint16_t          timeout )
```

- Process plug-in **environment** configuration
 - covered later in the presentation
- Manage a map between URLs and plug-in factories:
 - ie. `root://eosatlas.cern.ch:1094` ▸ `XrdEosFactory`
 - factories are objects that instantiate plug-ins (think of: `new XrdEosFile`) given URLs

- **File object creation** (constructor)
 - ask the plug-in manager whether a plug-in for a given URL is known
 - if so, install the plug-in
- **File object usage** (method calls)
 - call the plug-in if it is installed
 - call the normal XRootD code if there is no plug-in present

```
]==> cat eos.conf  
# example configuration  
  
url = eosatlas.cern.ch;eoscms.cern.ch  
lib = /usr/lib64/libXrdEosClient.so  
enable = true
```

- The plug-ins are discovered and configured by scanning configuration files
- There is one config file per plug-in
- It's a set of key value pairs

- The plug-in manager will search for global configuration files in:

```
/etc/xrootd/client.plugins.d/
```


- The plug-in manager will search for global configuration files in:

```
/etc/xrootd/client.plugins.d/
```

- The global settings may be overridden by configuration files found in:

```
~/ .xrootd/client.plugins.d/
```

- The plug-in manager will search for global configuration files in:

```
/etc/xrootd/client.plugins.d/
```

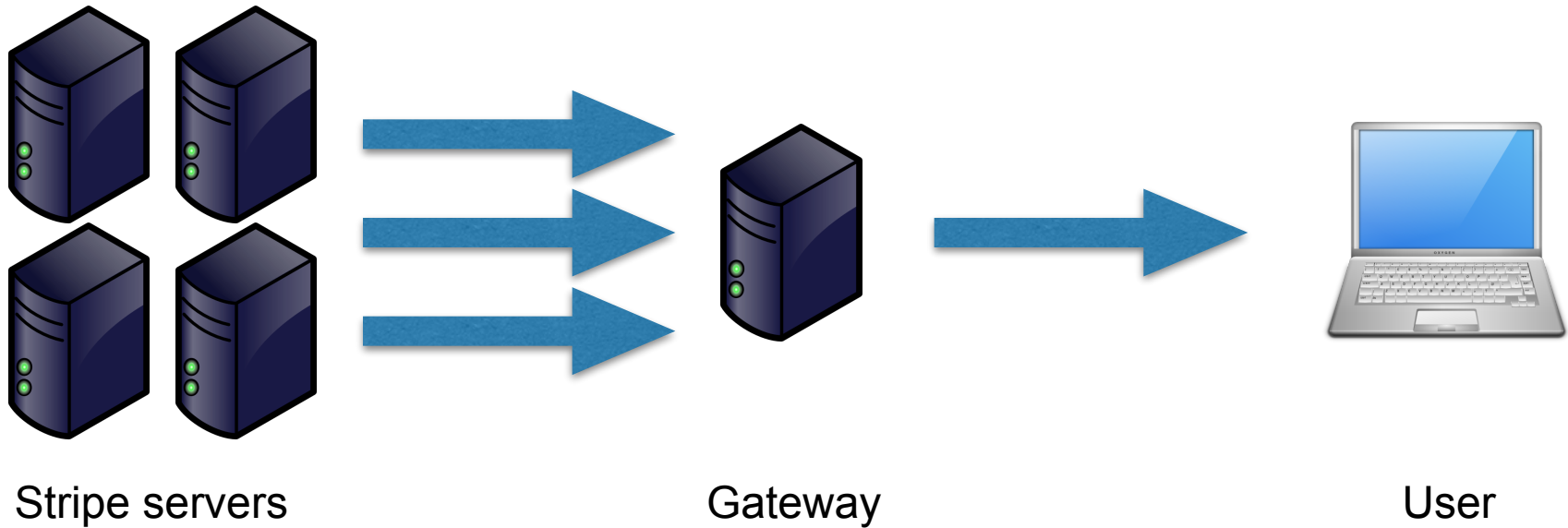
- The global settings may be overridden by configuration files found in:

```
~/ .xrootd/client.plugins.d/
```

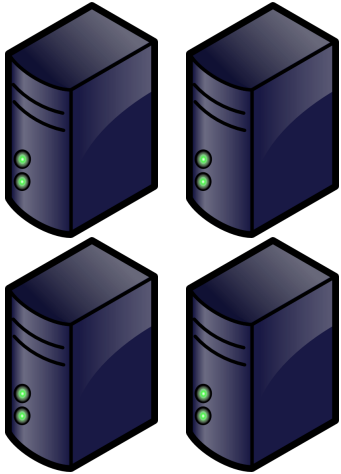
- Any of the previous settings may be overridden by configuration files found in a directory pointed to by:

```
XRD_PLUGINCONFDIR
```

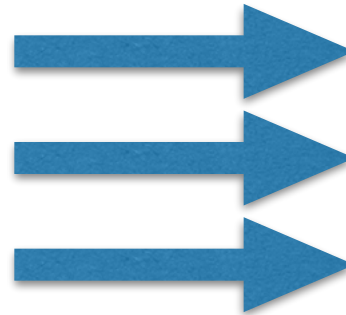
- Plug-ins may be developed and distributed **independently** of the XRootD code
- The plug-in manager performs strict interface **version checking**
 - will refuse to load ABI incompatible plug-ins
- Plug-in package (RPM) needs to contain:
 - the plug-in shared library
 - a config file in `/etc/xrootd/client.plugins.d/`



- Primary motivation for the plugin
- The client needs to see the file as a whole
- File reconstruction needs to be done at a gateway
- CPU and bandwidth scalability issues



Stripe servers



User

- When contacting EOS the client is able to execute specialised code
- Can contact the stripe servers directly
- Can reconstruct the data at the client machine
- **Transparently to the user - whatever he is!**

- The plug-in writer can re-implement all the File and FileSystem calls, so all sorts of things can be done:
 - access files using other protocols (http://, ceph://, ...)
 - opening multiple files at the same time and doing CMS-style client-side dynamic load balancing (ie. in the federation context)
 - handle cross protocol redirections
 - endless possibilities
- Enable the community to tinker, exactly what made the XRootD server successful
- And all this transparent to the layers above!

- Fully functional
- Available for testing in XRootD GitHub repo master branch
- Will be released with **XRootD 4.0.0**

- The plug-ins are:
 - **flexible** - override all possible calls, do whatever you want
 - **independent** - development and deployment may be completely detached from XRootD core
 - **ready to test** in the master branch
 - give a possibility to freely **experiment** and then incorporate new things into the core package

Thanks for your attention!

Questions? Comments?