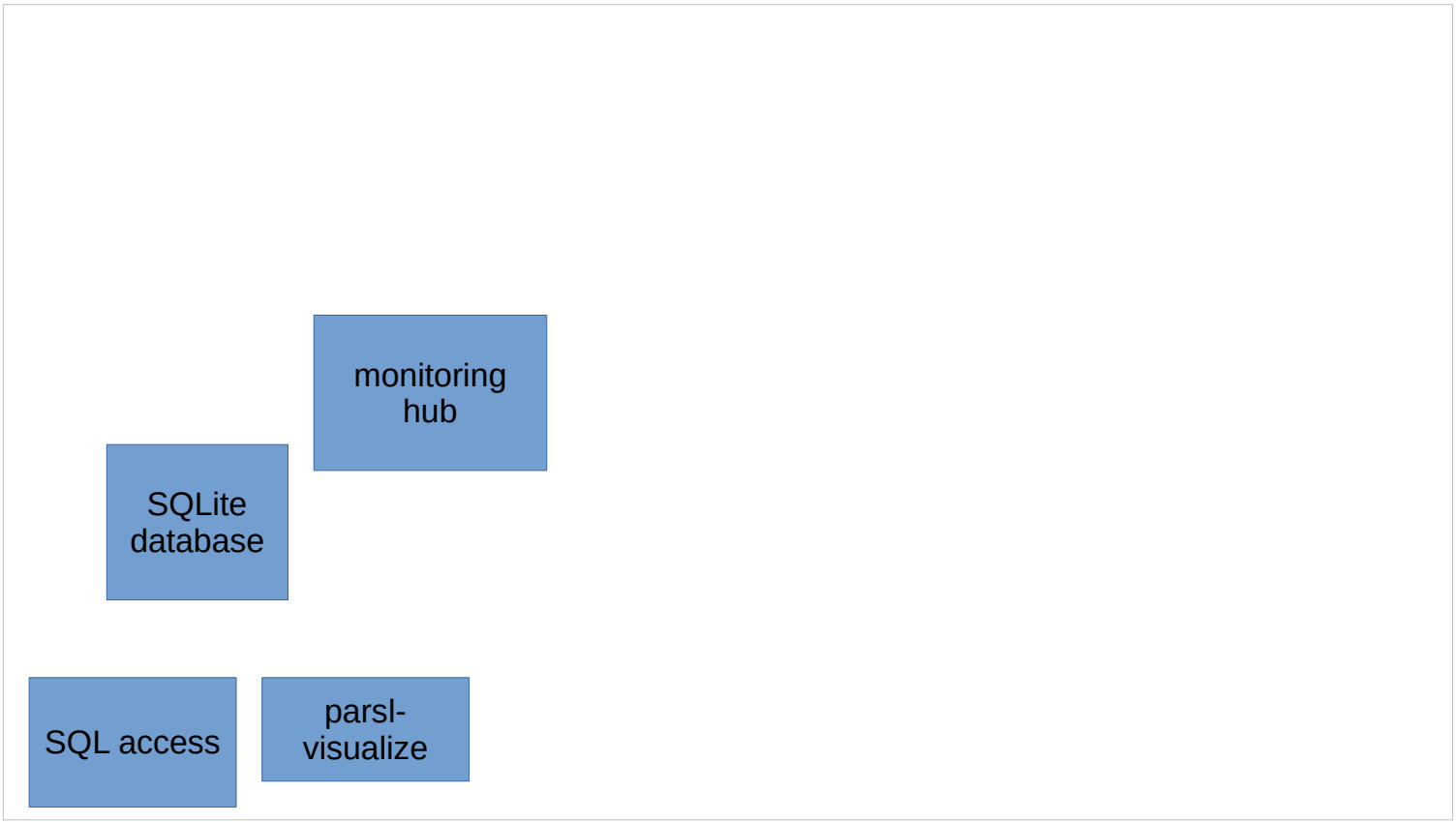
Parsl Monitoring Message Flow

Ben Clifford

ParslFest 2025

August 2025



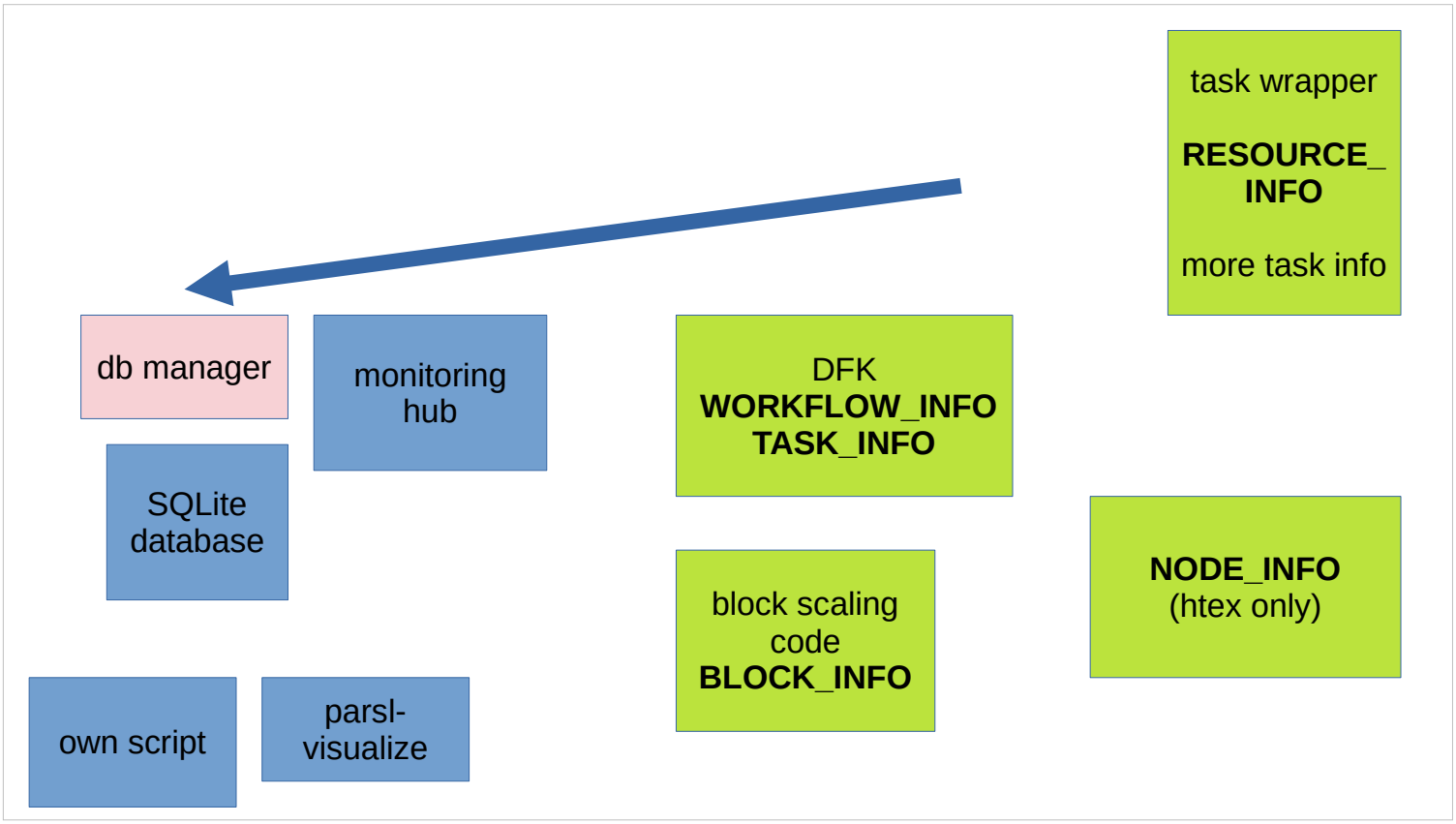
As a user you interact with monitoring with these components:

you configure the MonitoringHub object, and you get an sqlite3 database, which is a file on disk, with data about your workflow runs.

and you interact with it either using parsl-visualize or by your own SQL code.

This talk is not about that - this talk is how data flows into the database in the first place.

and on how you might fiddle with that flow, as part of work I've been doing to make Parsl more modular and hackable.



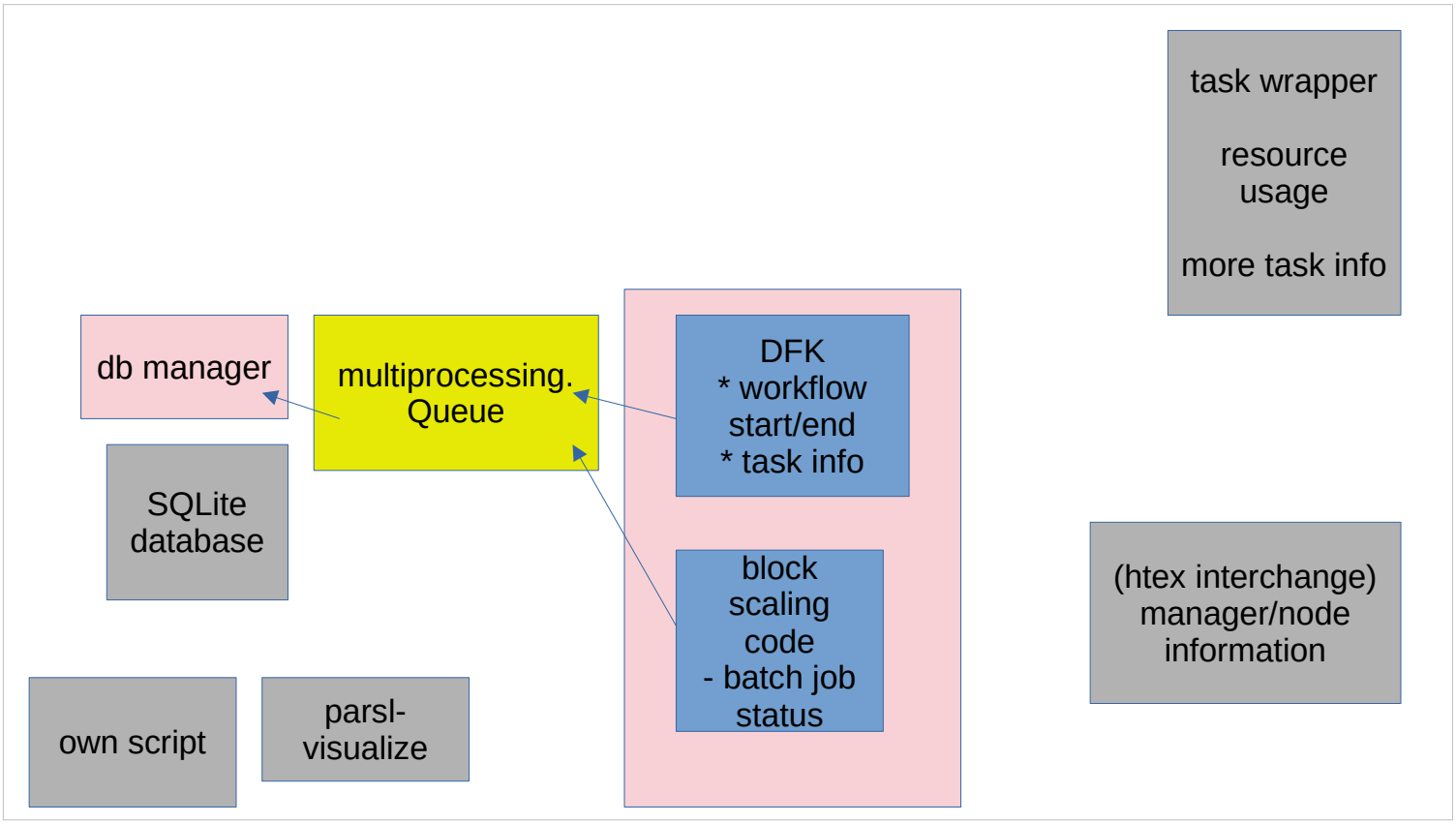
The only process which writes to this sqlite3 database is the "database manager" which sits alongside your submit side workflow.

It receives *messages* from various components of Parsl saying what is happening with that component. These messages roughly but not exactly translate into rows in the monitoring database.

TASK_INFO
RESOURCE_INFO (which is actually maybe two types?)
WORKFLOW_INFO
NODE_INFO
BLOCK_INFO

What monitoring messages look like is:
there are various sources of monitoring messages
(especially highlight which parts are also present in globus compute - to help GC devs)

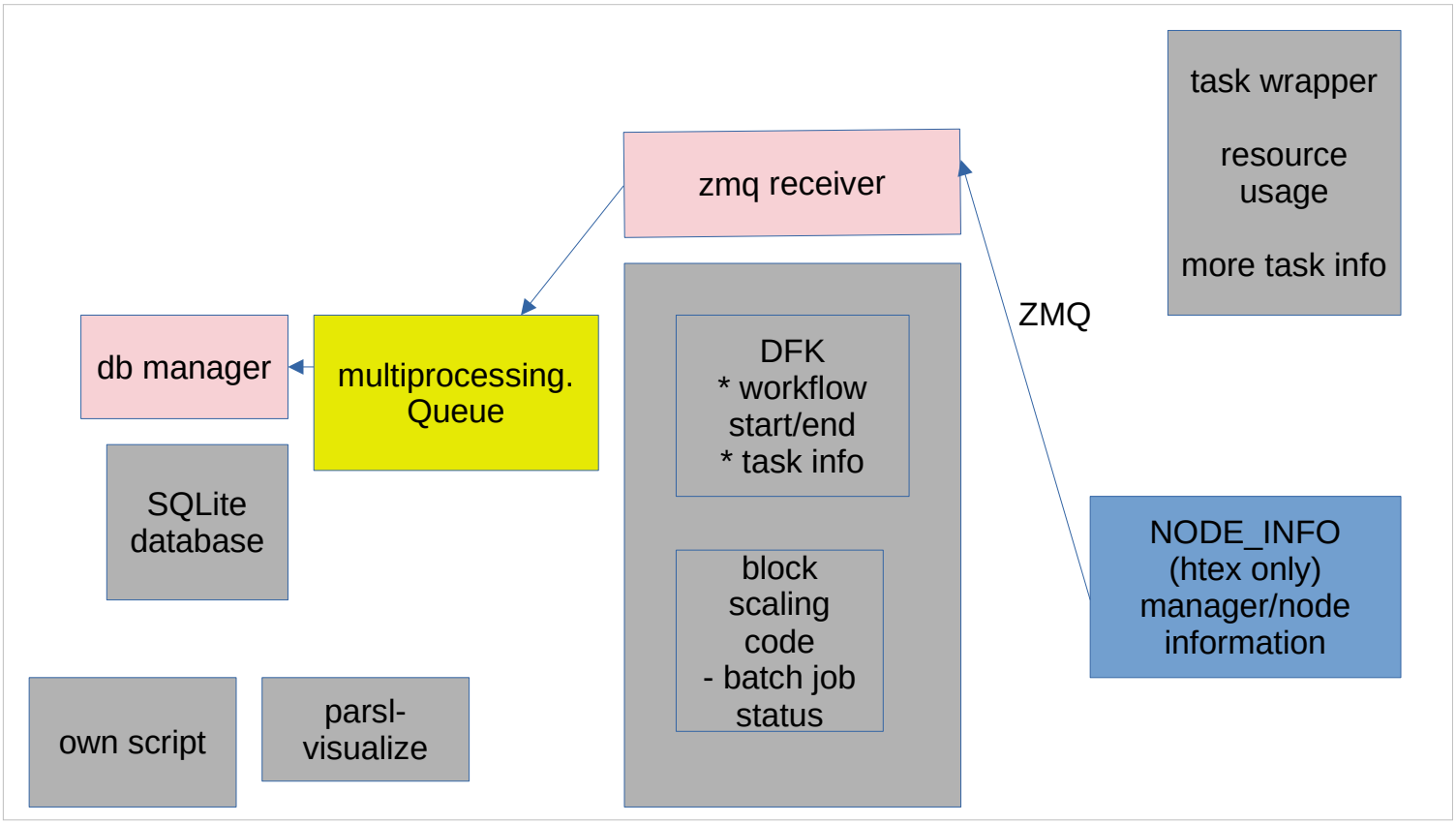
RESOURCE_INFO is actually two different kinds of message with the same tag - would make sense to separate them [suggestion]



now I’m going to talk about the protocols that are used to get messages from these components to the database manager process.

closest to the database manager - the relevent bit of the monitoring hub here is that it sets up a shared Python multiprocessing queue. Any process in a multiprocessing group can put messages into that queue; and the database manager takes them out and processes them

so to begin with, all the messages from your main workflow submit side process are sent that way - thats these ones in this process box - workstart start/end, block information, task_info



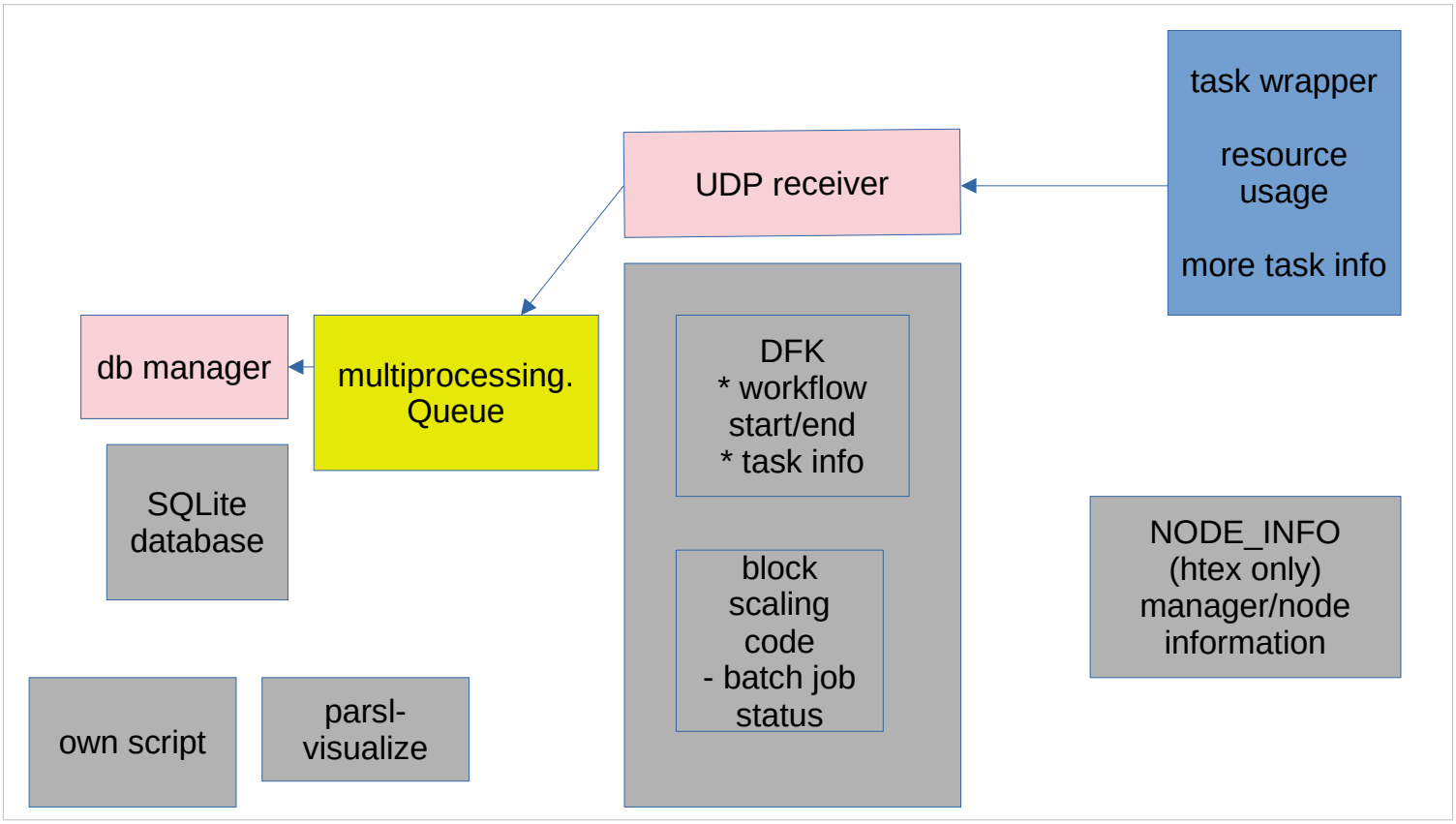
what about the other processes that aren't in the multiprocessing group? the processes running as task wrappers aren't even on the same computer - they're on worker nodes. so there's going to have to be some kind of networking involved. and the interchange is a completely separate unix process (for historical reasons) even though its on the same host.

for this, there's an abstraction called "monitoring radios" - nothing too fancy. Different classes implement different protocols, but the basic model is:

on one end, Parsl starts up a receiver in the multiprocessing group, that receives messages and sticks them into that multiprocessing.Queue - it's a message forwarder/router.

on the other end, there's a sender that knows how to get messages to the receiver using that particular protocol.

the example I've added here is how the htex interchange can send its NODE_INFO messages over ZMQ to a receiver with then forwards it onwards into the database process. this probably doesn't even need to be its own process but could live as a thread in the submit process -- but history has made it this way.

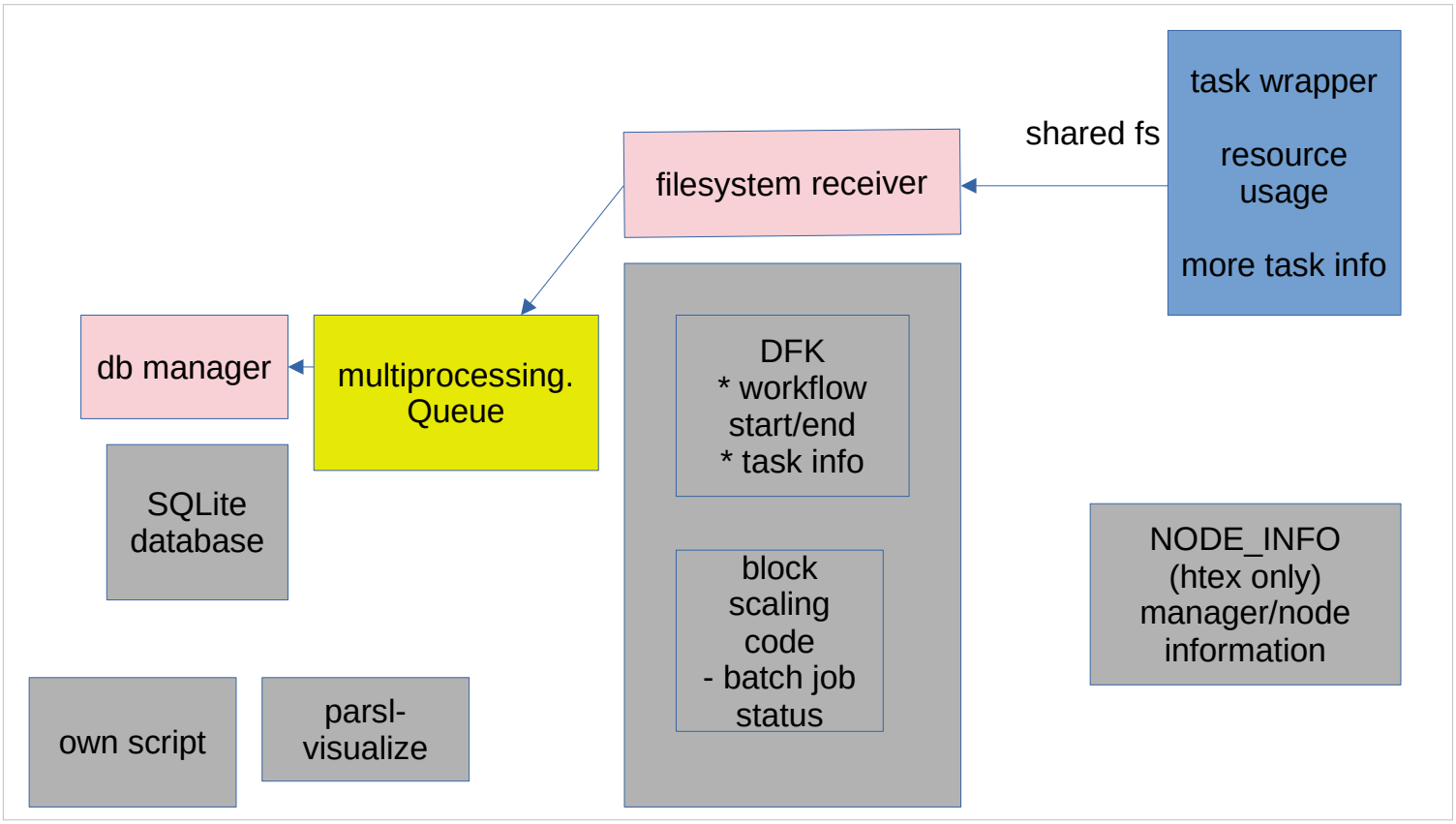


From the task wrapper, which is potentially running far away - the radio mechanism is pluggable. That's something I've worked on in the last year, with the intention that you can swap in different mechanisms here.

The original monitoring system used UDP - that's what this diagram is showing.

There's a UDP receiver process -- similar to the ZMQ one, it gateways messages from its own protocol into the multiprocessing.Queue.

That mechanism is quite flawed though: the monitoring database expects reliable message delivery, and UDP does not provide that. This isn't a theoretical problem in some networking lecture -- it's something I experienced with real applications.

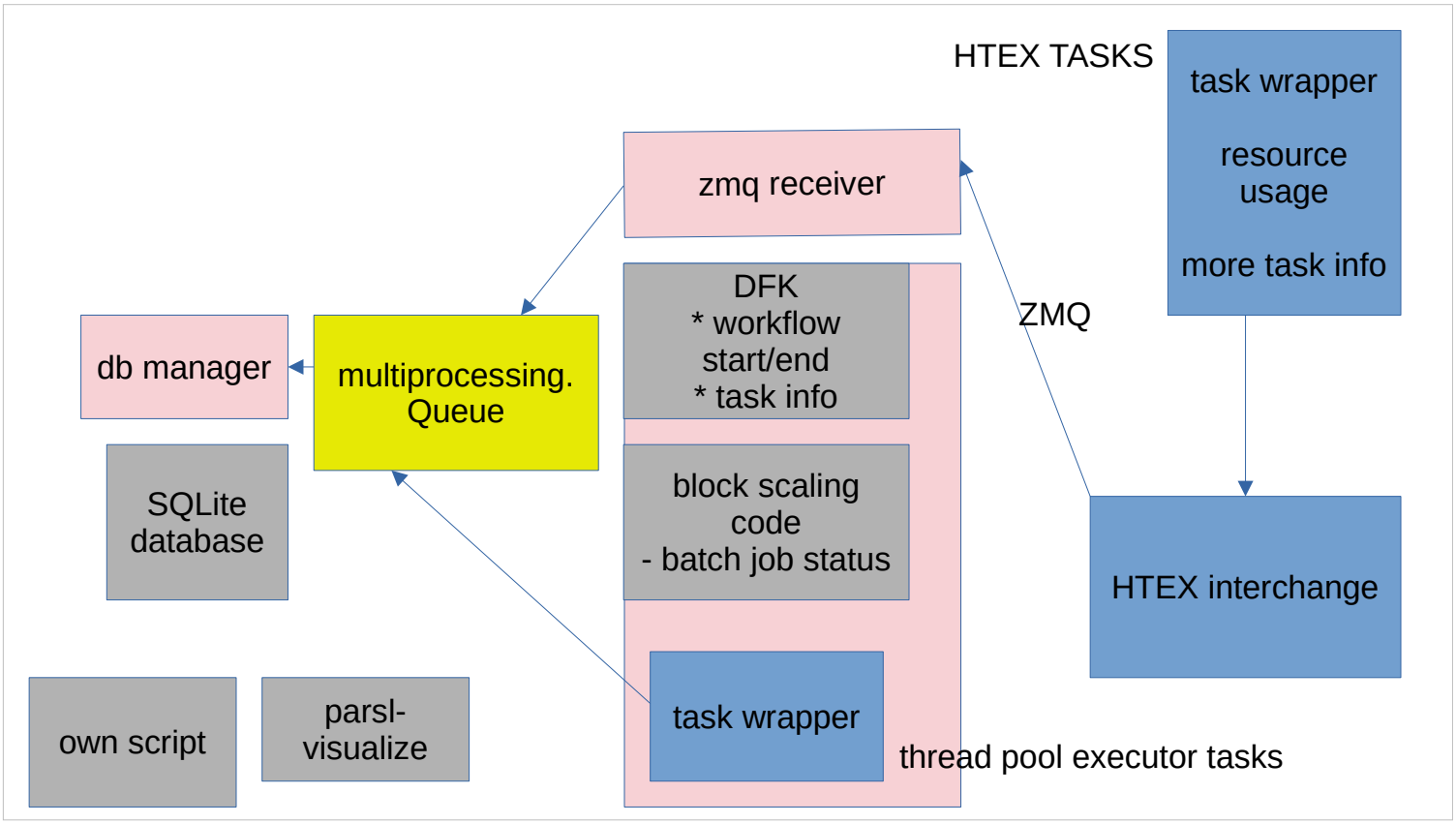


so the other option, which most executors have used by default in recent years, is the filesystem radio. That uses a shared filesystem directory: messages are written to the filesystem by the sender, and read out of the directory and fed into the queue by a receiver process.

That's still a network protocol - but using the word "shared" to mean "network".

This doesn't perform very well - it is a huge tradeoff from the very lightweight UDP to get reliability. But as of the middle of this year, you can easily choose between them.

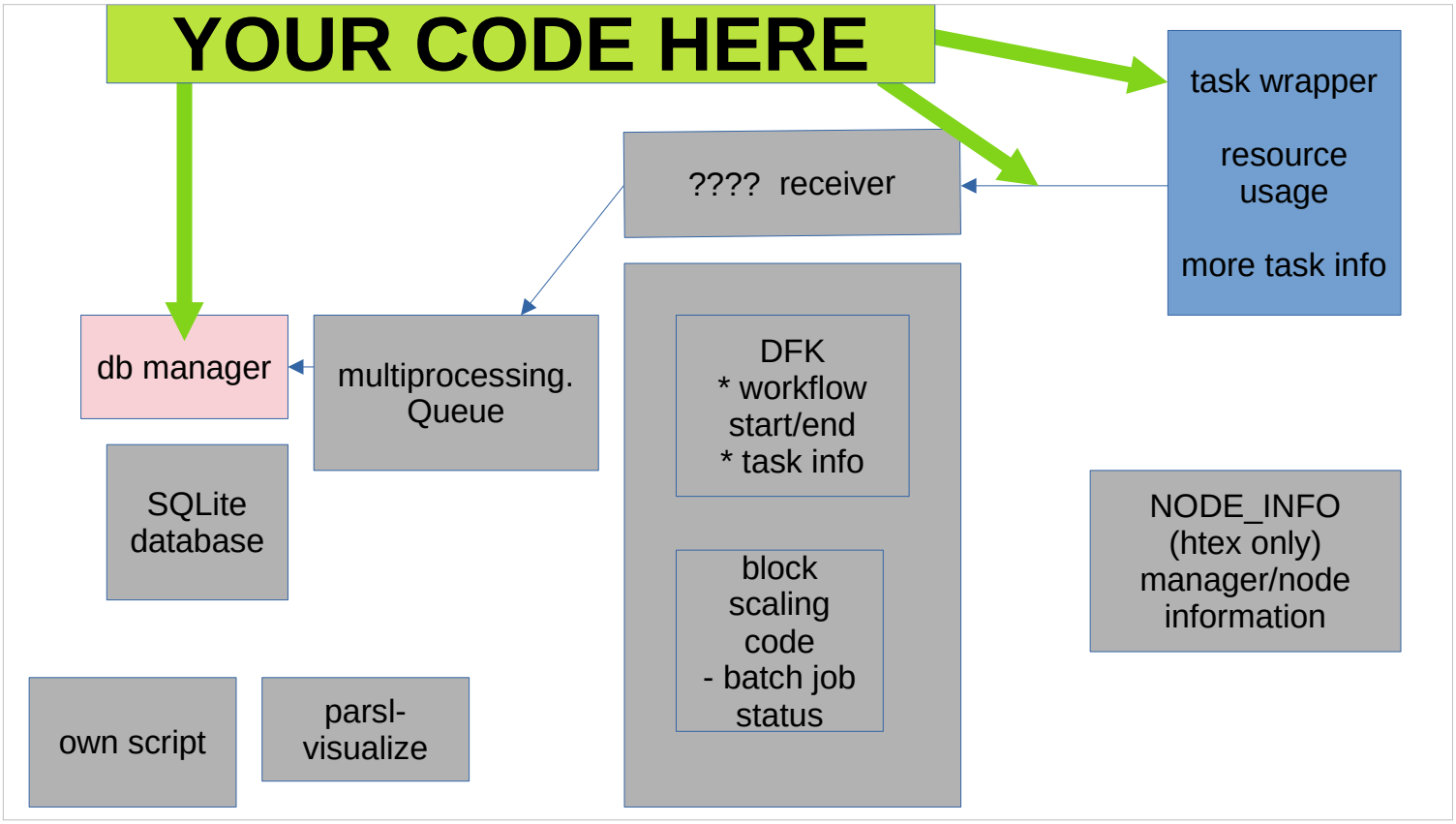
Or, work on something better and plug it in.



A couple of executors have their own special monitoring radios:

The thread executor runs tasks inside the submit side - which means it has direct access to the multiprocessing.Queue. There's no need for a receiver. The radio sender can write directly into the multiprocessing.Queue with no other infrastructure.

HTEX has a channel back as far as the interchange for results. That channel can be used for other stuff, like ... monitoring messages. And the interchange already has a channel to deliver messages into the multiprocessing.Queue, for NODE_INFO messages. So this radio piggy backs worker monitoring messages onto two channels that already exist.



I've shown one place where I think there is fun to be had plugging in different implementations - the monitoring radios getting messages from worker to the submit side.

There are a couple of other places that we've tossed ideas around for, but not made any production implementation:

* if you're running inside a bigger system, maybe you don't want a parsl-specific task information database - maybe you want your bigger system's task information system to be updated by Parsl monitoring messages. Swap out the database manager - for something that pulls messages off the queue and does something else with them.

* the task wrapper is quite inefficient. we've talked about having a per-node single process that reports on tasks. make the same RESOURCE_INFO messages, but send them from that per-node monitor.