Multiprocess Bitcoin

On unix systems, the `--enable-multiprocess` build option can be passed to `./configure` to build new `bitcoin-node`, `bitcoin-wallet`, and `bitcoin-gui` executables alongside existing `bitcoind` and `bitcoin-qt` executables.

`bitcoin-node` is a drop-in replacement for `bitcoind`, and `bitcoin-gui` is a drop-in replacement for `bitcoin-qt`, and there are no differences in use or external behavior between the new and old executables. But internally (after #10102), `bitcoin-gui` will spawn a `bitcoin-node` process to run P2P and RPC code, communicating with it across a socket pair, and `bitcoin-node` will spawn `bitcoin-wallet` to run wallet code, also communicating over a socket pair. This will let node, wallet, and GUI code run in separate address spaces for better isolation, and allow future improvements like being able to start and stop components independently on different machines and environments.

Next steps

Specific next steps after #10102 will be:

- [ ] Adding `-ipcbind` and `-ipcconnect` options to `bitcoin-node`, `bitcoin-wallet`, and `bitcoin-gui` executables so they can listen and connect to TCP ports and unix socket paths. This will allow separate processes to be started and stopped any time and connect to each other.
- [ ] Adding `-server` and `-rpcbind` options to the `bitcoin-wallet` executable so wallet processes can handle RPC requests directly without going through the node.
- [ ] Supporting windows, not just unix systems. The existing socket code is already cross-platform, so the only windows-specific code that needs to be written is code spawning a process and passing a socket descriptor. This can be implemented with `CreateProcess` and `WSADuplicateSocket`. Example: https://memset.wordpress.com/2010/10/13/win32-api-passing-socket-with-ipc-method/.
- [ ] Adding sandbox features, restricting subprocess access to resources and data. See https://eklitzke.org/multiprocess-bitcoin.

Debugging

The `-debug=ipc` command line option can be used to see requests and responses between processes.

Installation

The multiprocess feature requires Cap'n Proto and libmultiprocess as dependencies. A simple way to get starting using it without installing these dependencies manually is to use the depends system with the `MULTIPROCESS=1` dependency option passed to make:

```
cd <BITCOIN_SOURCE_DIRECTORY>
make -C depends NO_QT=1 MULTIPROCESS=1
CONFIG_SITE=$PWD/depends/x86_64-pc-linux-gnu/share/config.site ./configure
make
src/bitcoin-node -regtest -printtoconsole -debug=ipc
BITCOIND=bitcoin-node test/functional/test_runner.py
```

The configure script will pick up settings and library locations from the depends directory, so there is no need to pass `--enable-multiprocess` as a separate flag when using the depends system (it's controlled by the

`MULTIPROCESS=1` option).

Alternately, you can install Cap'n Proto and libmultiprocess packages on your system, and just run `./configure --enable-multiprocess` without using the depends system. The configure script will be able to locate the installed packages via pkg-config. See Installation section of the libmultiprocess readme for install steps. See build-unix.md and build-osx.md for information about installing dependencies in general.

IPC implementation details

Cross process Node, Wallet, and Chain interfaces are defined in `src/interfaces/`. These are C++ classes which follow conventions, like passing serializable arguments so they can be called from different processes, and making methods pure virtual so they can have proxy implementations that forward calls between processes.

When Wallet, Node, and Chain code is running in the same process, calling any interface method invokes the implementation directly. When code is running in different processes, calling an interface method invokes a proxy interface implementation that communicates with a remote process and invokes the real implementation in the remote process. The libmultiprocess code generation tool internally generates proxy client classes and proxy server classes for this purpose that are thin wrappers around Cap'n Proto client and server classes, which handle the actual serialization and socket communication.

As much as possible, calls between processes are meant to work the same as calls within a single process without adding limitations or requiring extra implementation effort. Processes communicate with each other by calling regular C++ interface methods. Method arguments and return values are automatically serialized and sent between processes. Object references and `std::function` arguments are automatically tracked and mapped to allow invoked code to call back into invoking code at any time, and there is a 1:1 threading model where any thread invoking a method in another process has a corresponding thread in the invoked process responsible for executing all method calls from the source thread, without blocking I/O or holding up another call, and using the same thread local variables, locks, and callbacks between calls. The forwarding, tracking, and threading is implemented inside the libmultiprocess library which has the design goal of making calls between processes look like calls in the same process to the extent possible.