Block and Transaction Broadcasting with ZeroMQ

ZeroMQ is a lightweight wrapper around TCP connections, inter-process communication, and shared-memory, providing various message-oriented semantics such as publish/subscribe, request/reply, and push/pull.

The Bitcoin Core daemon can be configured to act as a trusted "border router", implementing the bitcoin wire protocol and relay, making consensus decisions, maintaining the local blockchain database, broadcasting locally generated transactions into the network, and providing a queryable RPC interface to interact on a polled basis for requesting blockchain related data. However, there exists only a limited service to notify external software of events like the arrival of new blocks or transactions.

The ZeroMQ facility implements a notification interface through a set of specific notifiers. Currently there are notifiers that publish blocks and transactions. This read-only facility requires only the connection of a corresponding ZeroMQ subscriber port in receiving software; it is not authenticated nor is there any two-way protocol involvement. Therefore, subscribers should validate the received data since it may be out of date, incomplete or even invalid.

ZeroMQ sockets are self-connecting and self-healing; that is, connections made between two endpoints will be automatically restored after an outage, and either end may be freely started or stopped in any order.

Because ZeroMQ is message oriented, subscribers receive transactions and blocks all-at-once and do not need to implement any sort of buffering or reassembly.

Prerequisites

The ZeroMQ feature in Bitcoin Core requires the ZeroMQ API >= 4.0.0 libzmq. For version information, see dependencies.md. Typically, it is packaged by distributions as something like *libzmq3-dev*. The C++ wrapper for ZeroMQ is *not* needed.

In order to run the example Python client scripts in the `contrib/zmq/` directory, one must also install PyZMQ (generally with `pip install pyzmq`), though this is not necessary for daemon operation.

Enabling

By default, the ZeroMQ feature is automatically compiled in if the necessary prerequisites are found. To disable, use –disable-zmq during the *configure* step of building bitcoind:

```
$ ./configure --disable-zmq (other options)
```

To actually enable operation, one must set the appropriate options on the command line or in the configuration file.

Usage

Currently, the following notifications are supported:

```
-zmqpubhashtx=address
-zmqpubhashblock=address
-zmqpubrawblock=address
-zmqpubrawtx=address
-zmqpubsequence=address
```

The socket type is PUB and the address must be a valid ZeroMQ socket address. The same address can be used in more than one notification. The same notification can be specified more than once.

The option to set the PUB socket's outbound message high water mark (SNDHWM) may be set individually for each notification:

```
-zmqpubhashtxhwm=n
-zmqpubhashblockhwm=n
-zmqpubrawblockhwm=n
-zmqpubrawtxhwm=n
-zmqpubsequencehwm=address
```

The high water mark value must be an integer greater than or equal to 0.

For instance:

```
$ bitcoind -zmqpubhashtx=tcp://127.0.0.1:28332 \
           -zmqpubhashtx=tcp://192.168.1.2:28332 \
           -zmqpubhashblock="tcp://[::1]:28333" \
           -zmqpubrawtx=ipc:///tmp/bitcoind.tx.raw \
           -zmqpubhashtxhwm=10000
```

Each PUB notification has a topic and body, where the header corresponds to the notification type. For instance, for the notification `-zmqpubhashtx` the topic is `hashtx` (no null terminator). These options can also be provided in bitcoin.conf.

The topics are:

`sequence`: the body is structured as the following based on the type of message:

```
<32-byte hash>C :                  Blockhash connected
<32-byte hash>D :                  Blockhash disconnected
<32-byte hash>R<8-byte LE uint> : Transactionhash removed from mempool for non-block inclusi
<32-byte hash>A<8-byte LE uint> : Transactionhash added mempool
```

Where the 8-byte uints correspond to the mempool sequence number.

`rawtx`: Notifies about all transactions, both when they are added to mempool or when a new block arrives. This means a transaction could be published multiple times. First, when it enters the mempool and then again in each block that

includes it. The messages are ZMQ multipart messages with three parts. The first part is the topic (`rawtx`), the second part is the serialized transaction, and the last part is a sequence number (representing the message count to detect lost messages).

```
| rawtx | <serialized transaction> | <uint32 sequence number in Little Endian>
```

`hashtx`: Notifies about all transactions, both when they are added to mempool or when a new block arrives. This means a transaction could be published multiple times. First, when it enters the mempool and then again in each block that includes it. The messages are ZMQ multipart messages with three parts. The first part is the topic (`hashtx`), the second part is the 32-byte transaction hash, and the last part is a sequence number (representing the message count to detect lost messages).

```
| hashtx | <32-byte transaction hash in Little Endian> | <uint32 sequence number in Little E
```

`rawblock`: Notifies when the chain tip is updated. Messages are ZMQ multipart messages with three parts. The first part is the topic (`rawblock`), the second part is the serialized block, and the last part is a sequence number (representing the message count to detect lost messages).

```
| rawblock | <serialized block> | <uint32 sequence number in Little Endian>
```

`hashblock`: Notifies when the chain tip is updated. Messages are ZMQ multipart messages with three parts. The first part is the topic (`hashblock`), the second part is the 32-byte block hash, and the last part is a sequence number (representing the message count to detect lost messages).

```
| hashblock | <32-byte block hash in Little Endian> | <uint32 sequence number in Little Endi
```

NOTE: Note that the 32-byte hashes are in Little Endian and not in the Big Endian format that the RPC interface and block explorers use to display transaction and block hashes.

ZeroMQ endpoint specifiers for TCP (and others) are documented in the ZeroMQ API.

Client side, then, the ZeroMQ subscriber socket must have the ZMQ_SUBSCRIBE option set to one or either of these prefixes (for instance, just `hash`); without doing so will result in no messages arriving. Please see `contrib/zmq/zmq_sub.py` for a working example.

The ZMQ_PUB socket's ZMQ_TCP_KEEPALIVE option is enabled. This means that the underlying SO_KEEPALIVE option is enabled when using a TCP transport. The effective TCP keepalive values are managed through the underlying operating system configuration and must be configured prior to connection establishment.

For example, when running on GNU/Linux, one might use the following to lower the keepalive setting to 10 minutes:

sudo sysctl -w net.ipv4.tcp_keepalive_time=600

Setting the keepalive values appropriately for your operating environment may improve connectivity in situations where long-lived connections are silently dropped by network middle boxes.

Also, the socket's ZMQ_IPV6 option is enabled to accept connections from IPv6 hosts as well. If needed, this option has to be set on the client side too.

Remarks

From the perspective of bitcoind, the ZeroMQ socket is write-only; PUB sockets don't even have a read function. Thus, there is no state introduced into bitcoind directly. Furthermore, no information is broadcast that wasn't already received from the public P2P network.

No authentication or authorization is done on connecting clients; it is assumed that the ZeroMQ port is exposed only to trusted entities, using other means such as firewalling.

Note that for `*block` topics, when the block chain tip changes, a reorganisation may occur and just the tip will be notified. It is up to the subscriber to retrieve the chain from the last known block to the new tip. Also note that no notification will occur if the tip was in the active chain–as would be the case after calling invalidateblock RPC. In contrast, the `sequence` topic publishes all block connections and disconnections.

There are several possibilities that ZMQ notification can get lost during transmission depending on the communication type you are using. Bitcoind appends an up-counting sequence number to each notification which allows listeners to detect lost notifications.

The `sequence` topic refers specifically to the mempool sequence number, which is also published along with all mempool events. This is a different sequence value than in ZMQ itself in order to allow a total ordering of mempool events to be constructed.