Satoshi's Original Bitcoin Client - An Operational View  
  
Preface  
---------  
  
I thought my client was taking too long to download the block chain and it  
did not appear to operate smoothly. I thought I could do something to decrease  
the block download time. So I downloaded the code and dug in.  Ultimately,  
I failed to find the silver bullet to eliminate the long download delays  
(big suprise!). But I did manage to penetrate the C++ code and figure  
out how things worked for the most part.  
  
So, I decided to write down my understanding of the code from an operational  
perspective, to spare those who are not fluent in C++ from having to wade  
through the code, which is quite dense and bit of a chore to pick apart,  
when they really just want to know "how it works".  
  
My focus was initially on the block download process, but I decided to  
go ahead and cover all the major operational aspects I could (before losing  
interest Wink. I do think I found some areas for improvement, but that is not  
the point of these articles. I will try to make it clear when I am stating  
the facts versus when I am writing commentary.  
  
I intend these articles to go into the Wiki at some point but I also  
thought it would be useful to open topics in the forum in order to  
allow for review in case I made a mistake or missed something big,  
and for reference.  
  
  
Overview  
------------  
  
This series of articles will focus on how the Satoshi bitcoin client  
program operates, and less so on the protocol details and the rules  
for processing blocks and transactions.  
  
Satoshi's bitcoin client is a C++ program, so be sure to look for code in  
both the .cpp and the .h header files. Also, the program is multithreaded.  
This leads to some complexity and the use of certain code patterns to deal  
with concurrency that may be unfamiliar to many programmers. Also, the  
code is aggresive in the use of C++ constructs, so it will help to be  
fluent with map, multmap, set, string, vector, iostream, and templates.  
  
  
For information on how the bitcoin protocol works, see:  
    The original Satoshi whitepaper:  
        http://bitcoin.org/bitcoin.pdf  
    The articles on the bitcoin.it Wiki:   
        https://en.bitcoin.it/wiki/Category:Technical  
    With special mention of the protocol specification:  
       <https://en.bitcoin.it/wiki/Protocol_specification>  
    And the protocol rules:  
       <https://en.bitcoin.it/wiki/Protocol_rules>  
  
  
-- Operations --  
  
The client is oriented around several major operations, including:   
  
    Initialization and Startup  
        Upon startup, the client performs various initilization routines  
        including starting multiple threads to handle concurrent operations.  
  
    Node Discovery  
        The client uses various techniques find out about other bitcoin  
        nodes that may exist.  
  
    Node Connectivity  
        The client initiates and maintains connections to other nodes.  
  
    Sockets and Messages  
        The client processes messages from other nodes and sends  
        messages to other nodes using socket connections.  
      
    Block Exchange  
        Nodes advertise their inventory of blocks to each other and  
        exchange blocks to build block chains.  
  
    Transaction Exchange  
        Nodes exchange and relay transactions with each other.  
        The client associates transactions with bitcoin addresses in the  
        local wallet.  
  
    Wallet Services  
        The client can create transactions using the local wallet.  
        The client associates transactions with bitcoin addresses in the  
        local wallet. The client provides a service for managing  
        the local wallet.  
  
    RPC Interface  
        The client offers an JSON-RPC interface over HTTP over sockets  
        to perform various operational functions and to manage the local  
        wallet.  
  
    User Interface  
        The user interface code is scheduled to be superseded by bitcoin-qt.  
        Therefore, it is not covered in further detail.  
  
See their individual articles for more detail on each of these operations.  
  
  
-- fClient Mode --  
  
It is worth noting that there is code in the client to allow it to  
operate in a mode where it only downloads block headers.  
The implementation is intended to be used as a lightweight client mode which  
can operate without verifying and storing all blocks and transactions.  
  
This is controlled by the fClient variable in the code which is currently  
hard coded to false.  This is currently not considered to be finished code.  
  
This mode is known as fClient mode and the phrase Simplified Payment  
Verification (or SPV) mode has also been used to describe a lightweight  
client approach.  
  
  
  
-- Main Thread Level Functions --  
  
init.cpp:  
    main()  
    ExitTimeout  
    Shutdown  
net.cpp:  
    StartNode  
    ThreadGetMyExternalIP  
    ThreadMapPort  
    ThreadSocketHandler  
    ThreadOpenConnections  
    ThreadMessageHandler  
rpc.cpp:  
    ThreadRPCServer  
irc.cpp:  
    ThreadIRCSeed  
db.cpp:  
    ThreadFlushWalletDB  
ui.cpp:  
    ThreadDelayedRepaint  
    SendingDialogStartTransfer  
  
  
-- Significant Classes By File --  
  
net.cpp/.h:  
    CNode:  handes one socket connection  
    CInv  
    CAddress  
    CMessageHeader  
    CRequestTracker  
  
main.cpp/.h:  
    CDiskTxPos  
    CInPoint  
    COutPoint  
    CTxIn  
    CTxOut  
  
    CTransaction  
    CMerkleTx  
    CTxIndex  
  
    CBlock  
    CBlockIndex  
    CDiskBlockIndex  
    CBlockLocator  
  
    CAlert : CUnsignedAlert  
  
wallet.cpp/.h  
    CWallet : CKeyStore  
    CReserveKey  
    CWalletTx : CMerkleTx  
    CWalletKey  
    CAccount  
    CAccountingEntry  
  
db.cpp/.h:  
    CTxDB  
    CKeyPool  
    CWalletDB  
  
bignum.h  
    CBigNum  
  
util.h  
    CCriticalSection: used for thread contention  
  
  
--  
Search on "Satoshi Client Operation" for more articles in this series.  
  
Transaction Exchange : <https://bitcointalk.org/index.php?topic=41730.0>  
Block Exchange : <https://bitcointalk.org/index.php?topic=41729.0>  
Sockets and Messages : <https://bitcointalk.org/index.php?topic=41727.0>  
Node Connectivity : <https://bitcointalk.org/index.php?topic=41726.0>  
Node Discovery : <https://bitcointalk.org/index.php?topic=41722.0>  
Initialization and Thread Startup : <https://bitcointalk.org/index.php?topic=41719.0>

---------------------------------------------------------

Satoshi Client Operation: Transaction Exchange  
---------------------------------------------------------  
  
The Satoshi client advertises locally generated transactions and relays  
transactions from other nodes. This article describes the operations  
that deal with this exchange of transactions.  
  
See this article for more information on how transactions are validated:  
<https://en.bitcoin.it/wiki/Protocol_rules#.22tx.22_messages>  
  
  
-- Wallet Send --  
  
The client periodically calls SendMessages() (in main.cpp) which calls  
ResendWalletTransactions to send transactions generated locallly.  
This routine looks to see if there has been a new block since last time,  
and if so, and the local transaction are still not in a block,  
then the transactions are sent to all nodes.  
This is done only about every 30 minutes.  
  
Transactions are only rebroadcast if they have a timestamp at least  
5 minutes older than the last block was received. They are sorted  
and sent oldest first.[1]  
  
  
-- Periodic Advertisement --  
  
The client periodically calls SendMessages() (in main.cpp) which  
determines if a message should be sent to a remote node.  
For each message processing iteration, one node is chosen as the  
"trickle node".[2]  This node is the only one chosen to receive  
an "addr" message, if appropriate.[3]  
  
In the section for inventory, the client sends 1/4 of the transaction  
inventory, determined randomly [4], UNLESS they are the trickle node,  
in which case they receive ALL transactions.[5] Yes that seems reversed,  
but it is what it is. If the node is to receive 1/4 (not all), then the  
code also avoids sending any transactions that came from the local  
wallet.[6]  The comments indicate this is intended to increase privacy.  
  
  
  
-- Relay --  
  
When the client receives a transaction via a "tx" messages,  
it calls RelayMessage, which calls RelayInventory, which queues the  
inventory to be sent to all other nodes.[7]  
  
  
  
Footnotes  
----------  
  
 1. See CWallet::ResendWalletTransactions in wallet.cpp.  
 2. See:  
       pnodeTrickle = vNodesCopy[GetRand(vNodesCopy.size())];  
      ... and ...  
       SendMessages(pnode, pnode == pnodeTrickle);  
    in ThreadMessageHandler2() in net.cpp.  
 3. See:  
        //  
        // Message: addr  
        //  
        if (fSendTrickle)  
        {    
    in SendMessages() in main.cpp.  
 4. See:  
       bool fTrickleWait = ((hashRand & 3) != 0);  
    in SendMessages() in main.cpp.  
 5. See:  
        // trickle out tx inv to protect privacy  
        if (inv.type == MSG\_TX && !fSendTrickle)  
        {     
    in SendMessages() in main.cpp.  
 6. See:  
        // always trickle our own transactions  
        if (!fTrickleWait)  
        {     
            CWalletTx wtx;  
  
            if (GetTransaction(inv.hash, wtx))  
               if (wtx.fFromMe)  
                  fTrickleWait = true;  
        }  
    in SendMessages() in main.cpp.  
 7. Both RelayMessage and RelayInventory in net.h.  
  
  
--  
Search on "Satoshi Client Operation" for more articles in this series.

Satoshi Client Operation: Block Exchange  
-----------------------------------------------  
  
This article describes how blocks are exchanged between nodes.  
See this article for more information on how blocks are validated:  
<https://en.bitcoin.it/wiki/Protocol_rules#Blocks>  
  
Upon initial connection, if the connection was not inbound[1],  
or in other words, if the connection was initiated by the local node,  
the version message is queued for sending immediately. When the remote node  
receives the version message it replies with its own version message.[2]  
  
When a node receives a "version" message, it may send a "getblocks" request  
to the remote node if EITHER:  
   1. The node has never sent an initial getblocks request to any node yet.  
   2. Or, this is the only active node connection. Presumably the node had  
   zero connections prior to this connection, so maybe it was disconnected  
   for a long time. So, it will ask for blocks to catch up.  
  
The getblocks message contains multiple block hashes that the requesting  
node already possesses, in order to help the remote note find the latest  
common block between the nodes.  The list of hashes starts with the latest  
block and goes back ten and then doubles in an exponential progression  
until the genesis block is reached.[3]  Since both nodes are hard coded  
with the genesis block, they are guaranteed to a least start there.  
If that block does not match for some reason, no blocks are exchanged.  
  
  
-- Inventory Messages --  
  
Note that the node receiving the getblocks request does not actually send  
full blocks in response. The node sends an "inv" message containing just  
the hashes of the series of blocks that fit the request, which verifies  
that the node does indeed have blocks to send that the remote node does  
not have (but does not presume the remote node wants the full blocks yet).  
  
When the local node receives the "inv" message, it will request the actual  
blocks with a "getdata" message. See Below.  
  
But first, here is more detail how the remote node sends the "inv" message  
in response to the "getblocks" request sent by the local node. The remote  
node calls pFrom->PushInventory which is a method on the CNode instance that  
represents the node that requested the blocks (the local node in this  
walkthrough), and PushInventory adds the block hash to the vInventoryToSend  
variable of the CNode. The SendMessages function in main.cpp will take the  
inv items out of vInventoryToSend and add it to a vInv variable which  
means they are really ready for sending.[4]  
   - side node -  
   The reason for the seperate variable is that some inventory items  
   (transactions only right now) may be "trickled" to the remote node,  
   which means they may kept from from being sent right away.  
When the vInv variable fills up with 1000 entries, a message is queued  
with those 1000 entries and the loop continues.  At the end, any  
remaining entries are sent in a final "inv" message.  
  
When the local node receives the "inv" message, it will request the actual  
block with a "getdata" message. To be precise, the node calls pfrom->AskFor  
to request the block, and that method queues the request for the blocks in  
mapAskFor, and the multipurpose SendMessage() sends the "getdata" requests  
in batches of 1000 entries from the map.[5]  
  
The code attempts to limit redundant requests to every 2 minutes for the  
same block by using a map called mapAlreadyAskedFor to delay the message  
if necessary.[6]  
  
  
-- Block Batching --  
  
The responding node to a "getblocks" request attempts to limit the response  
to the requestor to 500 blocks.[7]  
  
However, in a peculiar twist, if the requestor appears to have diverged  
from the main branch, the node will send as many blocks as necessary to  
replace the entire bad chain of the requestor, from the lastest common block  
between the nodes, up to the last block the requestor has (on their bad main  
branch). That is in addition to the 500 "catch up" blocks for main branch  
updates that will also be sent.[8]  
  
Commentary:  
    This begs the question: what is the purpose of the 500 limit batch code?  
    It cannot be to DoS protect the responder from sending too much, because  
    the requestor can easily claim they are diverged from the main branch and  
    the responder will attempt to send the entire chain. It cannot be to  
    protect the client because if the poor client had been fooled into  
    receiving a long fake chain, then why further punish them by DoSsing  
    them with the entire chain?  
      
    I think a possible assumption was made in thinking that a node off the  
    main branch will never be able to catch up if we limit the number of  
    blocks we send them to a number below the number of blocks they are  
    behind.[C1] However, that should not be the case. Eventually the  
    requestor should receive the main chain blocks as the "continue mechanism"  
    (see below) proceeds on to further blocks.   
    C1. <https://github.com/bitcoin/bitcoin/commit/52f4cb48590a706caf7a492e8d94b85620d5cd33>  
  
Note that in addition to a flat limit on the number of blocks queued for  
sending, bitcoind also limits the total size of the blocks that are being  
queued. This is currently limited to half the send buffer size[9], which is  
10MB, for a limit of 5MB of queued blocks for send.[10]  
  
Commentary:  
    The 5MB send size limit is enough for about 200 blocks these  
    days (Aug2011). Batching based on send limits is what matters for  
    recent blocks since around block 127000 when the block size reached  
    10K and 500 blocks exceeds the size limit (10K\*500=5MB).  
    The dominance of the size limit for all but the early blocks raises  
    the question of whether the 500 block limit has enough relevance to  
    remain in the code.  
  
    It was relevant for a period of time. The size limit [C1] came along long  
    after the 500 block limit [C2] and I can imagine there was no pressing  
    need to remove Satoshi's earlier code. No need to go mucking around stuff  
    with unknown side effects, perhaps.  
    C1. <https://github.com/bitcoin/bitcoin/commit/497317453422611a077f7f195eb193d3bb597a9c>  
    C2. <https://github.com/bitcoin/bitcoin/commit/c5c7911dab8732861affbe66849a100da62f7464>  
  
-- Batch Continue Mechanism --  
  
When a node is finished sending a batch of block inventory, it records the  
hash of the last block in the batch.[11]  When the node receives a request  
for that full block, it realizes the remove node is done with the current  
batch and directly queues a special "inv" message (bypassing the normal  
SendMessage mechanism) with one block hash entry containing the  
latest block hash.[12] When the remote node receives that "inv" message,  
it will see that it does not have that block and it it will ask for that  
block as described above. However, this time when it receives the block  
and processes it, it will notice that it does not have the previous block,  
so it will record the latest block as an "orphan" block, and will request a  
block update starting with the latest block it has up to the block before  
the orphan [13], in order to fill in the gap. That goes out as a "getblocks"  
request and the whole batch process repeats itself.  
  
However, there is a twist. When the next batch finishes, the remote node  
sending the blocks will send the "inv" with latest block hash as usual, and  
the local node will notice it already has this block in the orphan block  
map this time and so it will skip requesting the block and directly ask  
for a block update.[14] This process will continue until the last  
block prior to the latest block is received. At the end of processing   
that block, it will notice there is an orphan that pointed to this block  
and will process the orphan block, (and any other orphans, recursively)  
thus completing the entire process.[15]  
  
  
-- Stall Recovery --  
  
If the batching processes is interrupted for some reason, such as the  
remote node failing to honor the "Batch Continue Mechanism" or if a   
disconnection occurs, there is a way for the process to restart. When  
a new block is solved and advertised around[16], any nodes that are  
behind will notice the new block in the "inv" and that will trigger it  
to request a "getblocks" update from the node that sent it the message.  
That will cause blocks to be sent starting from wherever in the block  
chain that the node that is behind is currently at.  
  
  
-- Long Orphan Chains --  
  
In various tests, it has proven relatively common (say more than one  
in ten) to discover nodes that are significantly behind on the block  
chain, probably because they are in the process of catching up as well.  
Since a well connected node will have at least 8 and up to dozens of  
connections, it is fairly likely that a new node will connect to  
another node that is also catching up.  
  
Nodes that are catching up will advertise the blocks they are processing,  
as they accept blocks into their main chain, to every other node.[16]  
While there is code to prevent advertising old blocks before a certain  
checkpoint, that code also has a clause that does advertise blocks to  
remote nodes if the block height is over the remote node's current best  
height minus 2000 blocks.[17] This appears to allow nodes to "help" other  
nodes catch up, even if they are both processing old blocks.  
  
These advertisements cause the local node to request those blocks  
from the remote node, which could be blocks well into the future compared  
to what has been processed locally. Due to the way blocks are requested,  
the remote node will send a large batch of blocks in response and will  
continue sending blocks to the local node until it reaches the end.  
Note that this is likely to occur at the same time the local node is  
downloading earlier blocks on the main chain from another node. That  
process may eventually catch up with the orphan chain and produce a   
very, very long operation to revalidate and connect up all the orphan  
blocks. Orphan chains over ten thousand blocks long, taking over an hour  
to process are possible.  
  
Therefore, two nodes talking to each other that are both catching up can  
lead to suboptimal interactions, especially when one both are far behind  
and one is far ahead of the other.  
  
  
-- Flood Limit Effects --  
  
Even with the batching mechanism described above, there are scenarios  
that occur that result in the remote node overflowing the local receive  
  
buffer while blocks are being exchanged.  
  
For example, if a remote node is "catching up", it will advertise each block  
it processes to the local node in certain circumstances (see above [17]).  
The local node will request each of those blocks right away. There is no  
protection against the local node requesting too many of these blocks.  
The remote node will send all blocks requested. There is no protection  
against the remote node sending too many blocks before the local node has  
time to process them, in this circumstance.  
  
The local receive buffer can fill up. When the local node notices a receive  
buffer is full, it disconnects that node connection.[18]  
If sets the fDisconnect flag, and once the buffers are empty[19], the  
socket is closed.  
  
  
-- Performance --  
  
As of September 1, 2011, on a server class computer circa 2005 running  
Ubuntu with a Comcast cable internet connection takes over 10 hours   
to download and process the block chain. While it is debatable what  
the bottleneck is early in the download process, it is clear from  
the processing of recent blocks that the network is not the bottleneck  
for all but the slowest internet connections.  
  
Blocks are taking over a second, on average, to process once downloaded.[20]  
However, the average size of a block is only around 24 kilobytes  
in August 2011. It certainly does not take 1 second to download 24K.  
Also, testing reveals very large queues of blocks being processed per  
message loop, which is not what you would expect if the thread was  
pulling them out of the queue as they arrive on the sockets.   
  
There are a number of "false signals" that lead one to believe the problem  
is with network performance. The first false signal is that, as of  
August 2011, nearly all of the first 60 or 70% of  blocks downloaded are  
very small. Recent average block sizes are around one hundred times bigger!  
So, almost all of a sudden, the block rate goes from very fast to very slow.  
It looks like something went wrong. In reality, if you measure the rate  
of block processing by kilobyte, the rate remains relatively constant.  
  
  
Another false signal is related to the fact that message queues are  
processed to completion, one at a time per node. This can result in big  
backups of messages from other nodes. So, a long period of increasing  
blocks may freeze for long periods as other nodes are serviced. Consider  
that block downloads typically come from just one remote node (at  
least until a miner or other relaying or downloading node advertises  
a late block and disrupts the process) and so all the work might  
be on one node. Things go fast processing the blocks from a node,  
and then that looks like it stops as "addr" messages are processed from  
other nodes and other work is done. But it looks like something is wrong.  
  
Also, the orphaning effects described above can lead to excessive block  
processing with nothing to show for it until the orphan chain is connected.  
Also, you do ocassionally run into a node that is slow to respond, perhaps  
because they are also processing blocks or because they have a slow machine  
or connection.  
  
All of the above contributes to heavy "jitter" in the block download process,  
and that is a more frustrating user experience than a constant download rate.  
  
  
Commentary:  
    The download delay has become much worse recently. Block sizes have  
    reached the point where it takes (using the once second avg) many  
    hours to process the chain and that time is increasing about an hour  
    every three weeks. So, if your laptop was off for three weeks, it  
    could take an hour to catch up. Initial download times are currently  
    set to increase 15 hours per year at todays transaction volume.  
    Clearly, downloading a preprocessed database of the block chain will  
    become an increasingly desirable option for many users.  
    Moreover, the current mechanism does not appear to be appropriate  
    for the casual user who runs the software part time on a laptop or home  
    computer, for example, and wants to have on-demand access to block  
    chain verification using the bitcoin protocol. Several solutions  
    proposed so far requires these users to place trust in something more  
    than the bitcoin network and protocol itself. Several "trust events" in  
    the bitcoin community in 2011 have probably hindered these solutions,  
    but in my opinion they are necessary.  
  
  
  
  
Footnotes  
 1. See pfrom->fInbound where pfrom is a CNode.  
 2. See ProcessMessage() in main.cpp where strCommand == "version".  
 3. See CBlockLocator in main.h.  
 4. See Message: inventory in SendMessage in main.cpp.  
 5. See Message: getdata at the end of SendMessage in main.cpp.  
 6. See CNode::AskFor in net.h.  
 7. See ProcessMessage() in main.cpp where strCommand =="getblocks".  
 8. See int nLimit = 500 + locator.GetDistanceBack();  
    in ProcessMessage in main.cpp where strCommand =="getblocks".  
 9. See if (--nLimit <= 0 || nBytes >= SendBufferSize()/2)  
    in ProcessMessage() in main.cpp where strCommand =="getblocks".  
10. See inline unsigned int SendBufferSize() {  
        return 1000\*GetArg("-maxsendbuffer", 10\*1000); }  
    in net.h.  
11. See pfrom->hashContinue = pindex->GetBlockHash();  
    in ProcessMessage() in main.cpp where strCommand =="getblocks".  
12. See: if (inv.hash == pfrom->hashContinue)  
    in ProcessMessage() in main.cpp where strCommand =="getdata".  
  
13. See:  
        // Ask this guy to fill in what we're missing  
        if (pfrom)  
            pfrom->PushGetBlocks(pindexBest, GetOrphanRoot(pblock2));  
    in ProcessBlock() in main.cpp.  
14. See:  
        else if (inv.type == MSG\_BLOCK && mapOrphanBlocks.count(inv.hash))  
            pfrom->PushGetBlocks(pindexBest, GetOrphanRoot(mapOrphanBlocks[inv.hash]));  
    in ProcessMessage() in main.cpp where strCommand =="inv".  
15. See:  
    // Recursively process any orphan blocks that depended on this one  
    in ProcessBlock() in main.cpp.  
16. See the last block of code in AcceptBlock in main.cpp.  
17. See:  
        if (nBestHeight > (pnode->nStartingHeight != -1 ? pnode->nStartingHeight - 2000 : 134444))  
    in AcceptBlock() in main.cpp.  
  
18. See: if (nPos > ReceiveBufferSize()) {  
    in ThreadSocketHandler2() in net.cpp.  
19. See:  
        if (pnode->fDisconnect ||  
            (pnode->GetRefCount() <= 0 && pnode->vRecv.empty() && pnode->vSend.empty()))  
    in ThreadSocketHandler2() in net.cpp.  
20. This is from the authors experience and also  
    see <https://bitcointalk.org/index.php?topic=31376.0>.  
  
Satoshi Client Operation: Sockets and Messages  
---------------------------------------------------------  
  
The original bitcoin client uses a multithreaded approach to socket  
handling and messages processing. There is one thread that handles  
socket communication (ThreadSocketHandler) and one (ThreadMessageHandler)  
which handles pulling messages off sockets and calling the  
processing routines. Both of these threads are in net.cpp.  
The message processing routines are in main.cpp, however.  
  
The socket thread reads the sockets and places data into a CDataStream  
associated with each node called vRecv. The Satoshi client uses C++  
serialization operators >> and << to read and write to a CDataStream  
and then it uses generic routines to move the data between the streams  
and sockets.  
  
The message thread reads and processes all messages from each node in  
sequence, and then it sends messages to each node that should be sent  
messages. That is all it does.  
  
Specifically, ThreadSocketHandler2 calls ProcessMessages(), which is  
located in main.cpp, to pull messages off each node's socket and  
process them. Then it calls SendMessages(), which is also located  
in main.cpp to create and send any messages appropriate for each node.  
  
ProcessMessages() attempts to find a message start signature in the  
vRecv stream. If it finds a message start, it deletes everything  
prior to the start. Then it reads the header, extracts the message  
type, and calls ProcessMessage on the message.  
  
SendMessages() actually creates and sends messages; it does not  
just send preexisting queued messages. It goes through various  
maps looking for work to do and produces a message and calls  
PushMessage method of CNode to send the message. PushMessage  
queues outbound data in the vSend data stream. (See PushMessage() in  
net.cpp). The socket thread handler then pulls data off the  
vSend data stream and calls send on the socket to send the data.  
  
Satoshi Client Operation: Node Connectivity  
---------------------------------------------------------------  
  
The Satoshi bitcoin client creates a thread to manage making  
connections to other nodes. The code for that thread is in a  
function called ThreadOpenConnections2 in net.cpp.  
  
The client also handles accepting new inbound connections and   
disconnecting nodes when appropriate in a a thread called  
ThreadSocketHandler2, which is also in net.cpp.  
  
The thread making connections does not discover the addresses of other  
nodes. That information is gathered in various ways (See the article  
on Node Discovery). The connection thread chooses among the available  
addresses and makes connections and disconnects nodes when appropriate.  
That is all it does.  
  
  
Node addresses are chosen based on the following set of rules.  
  
-- Outbound Static Addresses --  
  
If the user specified addresses with -connect, the node uses   
those addresses only. It tries to establish a connection to each node  
and then sleeps for a half second, and then repeats that in a loop  
until shut down. The code establishes a connection by calling  
OpenNetworkConnection(addr). If the connection is already open,  
OpenNetworkConnection just returns.  
  
  
If the user specified addresses with -node, then connections are  
made to those nodes (with a half second delay between each) upon  
startup. After those connections are attempted, the code proceeds  
to the regular connection handling code.  
  
  
-- Outbound Limiting --  
  
The connection handling code is one loop that performs various  
  
functions until shutdown. The first thing the loop does is count  
the number of outbound connections, and if the maximum has been  
reached (8 or -maxconnections), then it goes into a 2 second delay  
loop until the count is below the max.  
  
  
Assuming the number of connections is below the limit, the node attempts  
to connect to another node. See the next section.  
  
  
-- Seed Nodes --  
  
If the node has not been able to learn about other addresses, presumably  
because those methods have failed, the node will use an internal list  
of 320 node addresses hard coded into the software to populate  
the list of known node addresses.  
  
There is code to move away from seed nodes when possible. The presumption  
is that this is to avoid overloading seed nodes. Once the local node has  
enough addresses (presumably learned from the seed nodes), the  
connection thread will close seed node connections.  
  
-- Outbound Random Selection --  
  
First the code puts the addresses into a.b.c.\* buckets so only one  
connection is made to each 24 bit netmasked network.  
  
Next, it loops through every address and determines whether it is "ready",  
and then, using a complex calculation, computes a score for every address.  
The address with the highest score wins and OpenNetworkConnection is  
called for it. Then the code completes the main loop of the thread and  
continues.  
  
In order to determine readiness, the code hashes the IP and other entropy  
into a deterministic random number between 1 and 3600. If the address  
specifies a nonstandard port, a 2 hour (7200) penalty is added to the number.  
This is an adjustment number to the retry interval.  
  
The main retry interval is basically the square root of the last time seen  
  
plus the "random" adjustment from the previous paragraph. If the node  
has been seen in the last hour, however, the retry interval is set to  
ten minutes.  The following table is in the code:  
  
// Last seen  Base retry frequency  
//   <1 hour   10 min  
//    1 hour    1 hour  
//    4 hours   2 hours  
//   24 hours   5 hours  
//   48 hours   7 hours  
//    7 days   13 hours  
//   30 days   27 hours  
//   90 days   46 hours  
//  365 days   93 hours  
  
After computing the interval, if the address has already been contacted in  
the interval, the address is skipped.  
  
If the address is over a day old, we may skip it. If we are successfully  
getting IRC addresses, and have node connections, then we skip it with  
the assumption that we will see the address advertisement if it is really  
active.  
  
Finally, for all addresses that appear to be ready for a retry, the  
address that has not been contacted the longest is chosen with a maximum  
of 24 hours. However, there is a twist. The calculation for the score is this:  
int64 nScore = min(nSinceLastTry, (int64)24 \* 60 \* 60) - nSinceLastSeen - nRandomizer;  
So, the address is penalized for every second since it is last seen (and  
a random adjustment).  
  
Commentary:  
    The reason for the last seen penalty above is hard to understand.  
    I suppose it penalizes over advertised addresses, which might be good?  
  
  
  
-- Inbound Accepting and Disconnecting --  
  
The client handles accepting new inbound connections and disconnecting  
  
nodes when appropriate in a a thread called ThreadSocketHandler2,  
which is in net.cpp.  
  
The socket thread is simply a loop which disconnects sockets that  
have the fDisconnect flag set on them (and have empty buffers),  
prepares all sockets for "select" and calls "select". "select" is   
a system call which waits for activity on a set of sockets.  
When that call returns, the node accepts any new connections,  
receives and sends on any ready sockets, and marks any inactive sockets  
for disconnect with the fDisconnect flag.  
  
Sockets are disconnected if they are 60 seconds old and have not sent  
or received data.  
  
Sockets are disconnected if they have not sent or received data in  
the last 90 minutes.  
  
Sockets are disconnected if the current inbound data exceeds a buffer limit.  
(Search for: if (nPos > ReceiveBufferSize()) in net.cpp)  
  
Sockets are disconnected if the current outbound data exceeds a buffer limit.  
(Search for: if (vSend.size() > SendBufferSize()) in net.cpp)

Satoshi Client Operation: Node Discovery  
-------------------------------------------------  
  
The Satoshi client discovers the IP address and port of nodes in several  
different ways.  
  
1. Nodes discover their own external address by various methods.  
2. Nodes receive the callback address of remote nodes that connect to them.  
3. Nodes connect to IRC to receive addresses.  
4. Nodes makes DNS request to receive IP addresses.  
5. Nodes can use addresses hard coded into the software.  
6. Nodes exchange addresses with other nodes.  
7. Nodes store addresses in a database and read that database on startup.  
8. Nodes can be provided addresses as command line arguments  
9. Nodes read addresses from a user provided text file on startup  
  
A timestamp is kept for each address to keep track of when the node  
address was last seen. The AddressCurrentlyConnected in net.cpp handles  
updating the timestamp whenever a message is received from a node.  
Timestamps are only updated on an address and saved to the database  
when the timestamp is over 20 minutes old.  
  
See the Node Connectivity article for information on which type of  
addresses take precedence when actually connecting to nodes.  
  
In the first section we will cover how a node handles a request for  
addresses via the "getaddr" message. By understanding the role of  
timestamps, it will become more clear why timestamps are kept the way  
they are for each of the different ways an address is discovered.  
  
  
Handling Message "getaddr"  
-----------------------------------  
  
When a node receives a "getaddr" request, it first figures out how many  
addresses it has that have a timestamp in the last 3 hours.  
Then it sends those addresses, but if there are more than 2500 addresses  
seen in the last 3 hours, it selects around 2500 out of the available  
recent addresses by using random selection.  
  
  
Now lets look at the ways a node finds out about node addresses.  
  
  
1. Local Client's External Address  
-----------------------------------  
The client uses two methods to determine its own external, routable IP  
address: it uses IRC, preferably, and if that does not succeed, it uses  
public web services which return the information.  
  
From a thread created for this work (called ThreadIRCSeed in irc.cpp),  
the client makes an IRC connection to 92.243.23.21 or irc.lfnet.org,  
if the direct IP connection fails. The port is 6667.[1]  
If the connection succeeds, the client issues a USERHOST command to  
the IRC server, in order to get their own IP address.[2]  
  
The client also runs a thread called ThreadGetMyExternalIP (in net.cpp)  
which attempts to determine the client's IP address as seen from the outside  
world.  It gives the IRC thread a chance to discover the IP address  
first, sleeping and checking periodically for 2 minutes, and then it  
proceeds if the IRC method did not succeed.  
  
First, it attempts to connect to 91.198.22.70 port 80, which should be  
the checkip.dyndns.org server. If connection fails, a DNS request is made  
for checkip.dyndns.org and a connection is attempted to that address.  
Next, it attemps to connect to 74.208.43.192 port 80, which should be  
the [www.showmyip.com](http://www.showmyip.com/) server. If connection fails, a DNS request is made  
for [www.showmyip.com](http://www.showmyip.com/) and a connection is attempted to that address.  
  
For each address attempted above, the client attempts to connect,  
send a HTTP request, read the appropriate response line, and parse the  
IP address from it.  
If this succeeds, the IP is returned, it is advertised to any connected  
nodes, and then the thread finishes (without proceeding to the next  
address).  
  
  
  
2. Connect Callback Address  
  
-----------------------------------  
When a node receives an initial "version" message, and that node initiated  
the connection, then the node advertises its address to the remote so  
that it can connect back to the local node if it wants to.[3]  
After sending its own address, it sends a "getaddr" request message  
to the remote node to learn about more addresses, if the remote node  
version is recent or if the local node does not yet have 1000 addresses.  
  
  
3. IRC Addresses  
-----------------------------------  
  
In addition to learning and sharing its own address, the node  
learns about other node addresses via an IRC channel. See irc.cpp.  
After learning its own address, a node encodes its own address into a string  
to be used as a nickname. Then, it randomly joins an IRC channel named  
between #bitcoin00 and #bitcoin99. Then it issues a WHO command.  
The thread reads the lines as they appear in the channel and decodes  
the IP addresses of other nodes in the channel. It does this in  
a loop, forever, until the node is shutdown.  
  
When the client discovers an address from IRC, it sets the timestamp  
on the address to the current time, but it uses a "penalty"  
of 51 minutes, which means it looks like it was actually seen  
almost an hour ago.  
  
  
4. DNS Addresses  
-----------------------------------  
Upon startup, the client makes DNS requests to hard coded DNS names  
in order to learn about the addresses of other nodes.[4] As of version  
v0.3.24, these addresses were[5]:  
  
bitseed.xf2.org  
bitseed.bitcoin.org.uk  
dnsseed.bluematt.me  
  
A recent query of these addresses returned 48 nodes. Note that a  
DNS reply can contain multiple IP addresses for a requested name.  
  
  
Addresses discovered via DNS are initially given a zero timestamp,  
therefore they are not advertised in response to a "getaddr" request.  
  
  
  
5. Hard Coded "Seed" Addresses  
-----------------------------------  
The client contains hard coded IP addresses that represent bitcoin nodes.[6]  
These addresses are only used as a last resort, if no other method  
has produced any addresses at all.[7]  
When the loop in the connection handling thread ThreadOpenConnections2()  
sees an empty address map, it uses the "seed" IP addresses as backup.  
  
There is code is move away from seed nodes when possible. The presumption  
is that this is to avoid overloading those nodes. Once the local node has  
enough addresses (presumably learned from the seed nodes), the  
connection thread will close seed node connections.[8]  
  
Seed Addresses are initially given a zero timestamp,  
therefore they are not advertised in response to a "getaddr" request.  
  
  
  
6. Ongoing "addr" advertisements  
-----------------------------------  
Nodes may receive addresses in an "addr" message after having  
sent a "getaddr" request, or "addr" messages may arrive   
unsolicited, because nodes advertise addresses gratuitously  
when they relay addresses (see below), when they advertise  
their own address periodically, and when a connection is made.  
  
If the address is from a really old version, it is ignored; if from  
a not-so-old version, it is ignored if we have 1000 addresses already.  
  
If the sender sent over 1000 addresses, they are all ignored.  
  
Addresses received from an "addr" message have a timestamp, but the  
timestamp is not necessarily honored directly.  
  
  
For every address in the message:  
  \* If the timestamp is too low or too high, it is set to 5 days ago.  
  \* We subtract 2 hours from the timestamp and add the address.  
  
Note that when any address is added, for any reason, the code that calls  
AddAddress() does not check to see if it already exists. The AddAddresss()  
function in net.cpp will do that, and if the address already exists, further  
processing is done to update the address record. If the advertised services  
of the address have changed, that is updated and stored.  
If the address has been seen in the last 24 hours and the timestamp is  
currently over 60 minutes old, then it is updated to 60 minutes ago.  
If the address has NOT been seen in the last 24 hours, and the timestamp is  
currently over 24 hours old, then it is updated to 24 hours ago.  
  
-- Address Relay --  
  
Once addresses are added from an "addr" message (see above), they then  
may be relayed to the other nodes. First, the following criteria  
must be set [9]:  
    1. The address timestamp, after processing, is within 60 minutes  
    of the current time  
    2. The "addr" message contains 10 addresses or less  
    3. And fGetAddr is not set on the node. fGetAddr starts false,   
    is set to true when we request addresses from a node, and it is  
    cleared when we receive less than 1000 addresses from a node.  
    4. The address must be routable.  
When they meet the above criteria, the node hashes all the eligible  
node IP addresses, as well as the current day in the form of an integer,  
and the two nodes with the lowest hash value are chosen to have  
the address relayed to them.  
  
-- Self broadcast --  
  
Every 24 hours, the node advertises its own address to all connected nodes.  
It also clears the list of the addresses we think the remote node has, which  
will trigger a refresh of sends to nodes. This code is in SendMessages()  
in main.cpp.  
  
-- Old Address Cleanup --  
  
In SendMessages() in main.cpp, there is code to remove old addresses.  
This is done every ten minutes, as long as there are 3 active connections.  
The node erases messages that have not been used in 14 days as  
long as there are at least 1000 addresses in the map, and as long  
as the erasing process has not taken more than 20 seconds.  
  
  
  
7. Addresses stored in the Database  
-----------------------------------  
Addresses are stored in the database when AddAddress() is called.  
  
Addresses are read on startup when AppInit2() calls LoadAddresses(),  
which is located in db.cpp.  
  
Currently, it appears all addresses are stored all at once whenever  
any address is stored or updated [10]. Indeed, AddAddress is seen  
to take over .01 seconds in various testing and is typically called   
tens of thousands of times in the initial 12 hours of running the  
client.  
  
  
  
8. Command Line Provided Addresses  
-----------------------------------  
  
The user can specify nodes to connect to with the -addnode <ip>  
command line argument. Multiple nodes may be specified.  
  
Addresses provided on the command line are initially given a zero  
timestamp, therefore they are not advertised in response to a "getaddr"  
request.  
  
The user can also specify an address to connect to with the -connect <ip>  
command line argument. Multiple nodes may be specified.  
The -connect argument differs from -addnode in that -connect addresses  
are not added to the address database and when -connect is specified,  
  
only those addresses are used.  
  
  
  
9. Text File Provided Addresses  
-----------------------------------  
The client will automatically read a file named "addr.txt" in the  
bitcoin data directory and will add any addresses it finds in there  
as node addresses. These nodes are given no special preference over  
other addresses. They are just added to the pool.  
  
Addresses loaded from the text file are initially given a zero timestamp,  
therefore they are not advertised in response to a "getaddr" request.  
  
  
  
  
Footnotes:  
-----------------------------------  
 1. See: CAddress addrConnect("92.243.23.21", 6667); // irc.lfnet.org  
    in ThreadIRCSeed2() in irc.cpp.  
 2. See: GetIPFromIRC() in irc.cpp.  
 3. See: // Advertise our address  
    in ProcessMessage() in main.cpp where strCommand == "version"  
 4. See DNSAddressSeed() in net.cpp.  
 5. See "static const char \*strDNSSeed[] = {" in net.cpp  
 6. See pnSeed in net.cpp  
 7. See:  
        if (mapAddresses.empty() && (GetTime() - nStart > 60 || fTOR) && !fTestNet)  
    in ThreadOpenConnections2() in net.cpp.  
 8. See:  
            if (fSeedUsed && mapAddresses.size() > ARRAYLEN(pnSeed) + 100)  
            {     
                // Disconnect seed nodes  
    in ThreadOpenConnections2() in net.cpp.  
 9. See: if (addr.nTime > nSince && !pfrom->fGetAddr && vAddr.size() <= 10 && addr.IsRoutable())  
    in ProcessMessage() in main.cpp where strCommand == "addr"  
10. See <https://bitcointalk.org/index.php?topic=26436.0>

Satoshi Client Operation: Initialization and Thread Startup  
--------------------------------------------------------------------  
  
Note that the client uses a fixed number of threads that are created on  
startup. All of the reading and writing on active network sockets are  
handled by one thread.  
  
The main() entry point to the program is in init.cpp. Do not go looking in  
main.cpp for main() - you will not find it.  
  
The first thing main() does is call AppInit, which calls AppInit2 which  
performs a bunch of initialization busy work mostly related to how the  
application handles errors. At some point AppInit2 calls ParseParameters,  
and then some argument handling code processes some basic arguments.  
  
Now it checks to see if an existing GUI window is already present and if so,  
brings it to the foreground.  
  
Next it creates a lock file in the bitcoin directory and if bitcoin is already  
running, displays a message and exits.  
  
Next it attemps to listen on the bitcoin port, and if the port is already in  
user or other error, displays a message and exits.  
  
Next it loads IP addresses from a database, then it loads the block index, and  
then it loads the wallet.  
  
Next it gets the top block number from the wallet, or zero if -rescan is used.  
Then It scans the block chain from the block number above.  
  
Side Note:  
    Note that printf has been redefined in util.h. So printf is really  
    OutputDebugStringF which directs to a file if appropriate (see util.h  
    and util.cpp). Also "loop" is defined in util.h as for (  ; ;  ).  
  
Then it does a bit more initialization and parameter processing and then it  
finally opens the GUI main window.  
  
Next it creates two threads:  
  
    1. StartNode()  
    2. And if acting as server: ThreadRPCServer()  
This is the only place where these threads are created.  
  
StartNode() is in net.cpp and ThreadRPCServer() is in rpc.cpp.  
  
Last, if not compiled for the GUI it sleeps for 5 seonds in an infinite loop.  
  
  
  
-- Thread StartNode() in net.cpp --  
  
StartNode is sort of a master networking thread.  
  
First it creates a CNode for the localhost 127.0.0.1 internal addresss to  
handle communication.  
  
Next, it gets the local IP address. On windows, its a lookup using the  
local host name. On Linux, it is the first IP address it finds on an  
interface that is up and is not the loopback interface.  
  
Next it creates a thread (ThreadGetMyExternalIP) to confirm the local IP  
address using 3rd party verification. If using a proxy, it does not  
bother, because it will not take incoming connections.  
  
Next, if Universal Plug and Play is used, then a thread (ThreadMapPort)  
to deal with port mapping is created.  
  
Next it creates a thread (ThreadIRCSeed) to exchange IP addresses.  
  
Next it creates a thread (ThreadSocketHandler) to "Send and receive from  
sockets, accept connections".  
  
Next it creates a thread (ThreadOpenConnections) to initiate outbound  
connections.  
  
Next it creates a thread (ThreadMessageHandler) to process messages.  
  
Finally, if you have specified so, it will start a bitcoin mining thread.  
  
  
  
-- Thread IRCSeed in net.cpp --  
  
Connects to 92.243.23.21 port 6667, JOINs the channel, and then  
reads lines one at a time looking for other users.  
  
  
-- Thread ThreadSocketHandler in net.cpp --  
  
Goes into an endless loop servicing sockets that need servicing.  
Handles disconnected nodes.  
Prepares "select" descriptor list and then calls select, waiting for  
I/O on all the relevant sockets with a 50ms timeout.  
Process new incoming connection on listening socket. Create a CNode for them.  
Handle receiving and sending.  
Set sockets that have not done anything to disconnected state.  
  
  
-- Thread ThreadOpenConnecions in net.cpp --  
  
Figures out nodes from parameters, seeds, irc, etc, and then goes into a  
loop, connecting to each one by one.  
  
  
-- Thread ThreadMessageHandler in net.cpp  
  
This thread goes through all the nodes and calls ProcessMessages(pnode) in  
main.cpp which looks for valid messages on the node receive  
queue (pFrom->vRecv) and if it finds one, it calls   
ProcessMessage(CNode\* pfrom, string strCommand, CDataStream& vRecv), which  
is also in main.cpp.  
  
Then the thread calls SendMessages for each node (which is in main.cpp)  
which handles creating and sending any messages appropriate for that node.  
  
  
-- Thread ThreadRPCServer in rpc.cpp --  
  
  
This thread will be rather complicated in that it implements an  
HTTP(S)+JSONRPC server using boost classes that are probably going  
to be unfamiliar to most developers.  
  
You will see lines like this:  
    acceptor.set\_option(boost::asio::ip::tcp::acceptor::reuse\_address(true));  
  
and this:  
    boost::thread api\_caller(ReadHTTP, boost::ref(stream), boost::ref(mapHeaders), boost::ref(strRequest));  
    if (!api\_caller.timed\_join(boost::posix\_time::seconds(GetArg("-rpctimeout", 30))))  
  
It appears the code above is creating a thread in order to apply 30  
second timeout to reading an HTTPrequest.  When HTTP requests are  
read, they are parsed, and then a routine corresponding to the  
request command name is called to handle the request. This happens  
in a loop until shutdown.