Subject: **More BitCoin questions**

From: **Mike Hearn** <mike@plan99.net>
Date: Mon, Dec 27, 2010 at 8:21 PM
To: Satoshi Nakamoto <satoshin@gmx.com>

Happy Christmas Satoshi, assuming you celebrate it wherever you are in the world :-)

I have been working on a Java implementation of the simplified payment verification, with an eye to building a client that runs on Android phones. So I've been thinking a lot about storage requirements and the scalability of BitCoin, which led to some questions that the paper did not answer (maybe there could be a new version of the paper at some point, as I think aspects of it are now out of date).

Specifically, BitCoin has a variety of magic numbers and neither the code nor the paper explain where they came from. For example, the fact that inflation ceases when 21 million coins have been issued. This number must have been arrived at somehow, but I can't see how.

Another is the 10 minute block target. I understand this was chosen to allow transactions to propagate through the network. However existing large P2P networks like BGP can propagate new data worldwide in <1 minute.

The final number I'm interested in is the 500kb limit on block sizes. According to Wikipedia, Visa alone processed 62 billion transactions in 2009. Dividing through we get an average of 2000 transactions per second, so peak rate is probably around double that at 4000 transactions/sec. With a ten minute block target, at peak a block might need to contain 2.4 million transactions, which just won't fit into 500kb. Is this 500kb a temporary limitation that will be slowly removed over time from the official client or something more fundamental?

From: **Satoshi Nakamoto** <satoshin@gmx.com>
Date: Wed, Dec 29, 2010 at 10:42 PM
To: Mike Hearn <mike@plan99.net>

I have been working on a Java implementation of the simplified payment verification, with an eye to building a client that runs on Android phones. So I've been thinking a lot about storage requirements and the scalability of BitCoin, which led to some questions that the paper did not answer (maybe there could be a new version of the paper at some point, as I think aspects of it are now out of date).

The simplified payment verification in the paper imagined you would receive transactions directly, as with sending to IP address which nobody uses, or a node would index all transactions by public key and you could download them like downloading mail from a mail server.

Instead, I think client-only nodes should receive full blocks so they can scan them for their own transactions. They don't need to store them or index them. For the initial download, they only need to download headers, since there couldn't be any payments before the first time the program was run (a header download command was added in 0.3.18). From then on, they download full blocks (but only store the headers).

Code for client-only mode is mostly implemented. There's a feature branch on github with it, also I'm attaching the patch to this message.

Here's some more about it:

"Here's my client-mode implementation so far. Client-only mode only records block headers and doesn't use the tx index. It can't generate, but it can still send and receive transactions. It's not fully finished for use by end-users, but it doesn't matter because it's a complete no-op if fClient is not enabled. At this point it's mainly documentation showing the cut-lines for client-only re-implementers.

With fClient=true, I've only tested the header-only initial download.

A little background. CBlockIndex contains all the information of the block header, so to operate with headers only, I just maintain the CBlockIndex structure as usual. The nFile/nBlockPos are null, since the full block is not recorded on disk.

The code to gracefully switch between client-mode on/off without deleting blk*.dat in between is not implemented yet. It would mostly be a matter of having non-client LoadBlockIndex ignore block index entries with null block pos. That would make it re-download those as full blocks. Switching back to client-mode is no problem, it doesn't mind if the full blocks are there.

If the initial block download becomes too long, we'll want client mode as an option so new users can get running quickly. With graceful switch-off of client mode, they can later turn off client mode and have it download the full blocks if they want to start generating. They should rather just use a getwork miner to join a pool instead.

Client-only re-implementations would not need to implement EvalScript at all, or at most just implement the five ops used by the standard transaction templates."

Specifically, BitCoin has a variety of magic numbers and neither the
code nor the paper explain where they came from. For example, the fact
that inflation ceases when 21 million coins have been issued. This
number must have been arrived at somehow, but I can't see how.

Educated guess, and the maths work out to round numbers. I wanted something that would be not too low if it was very popular and not too high if it wasn't.

Another is the 10 minute block target. I understand this was chosen to
allow transactions to propagate through the network. However existing
large P2P networks like BGP can propagate new data worldwide in <1
minute.

If propagation is 1 minute, then 10 minutes was a good guess. Then nodes are only losing 10% of their work (1 minute/10 minutes). If the CPU time wasted by latency was a more significant share, there may be weaknesses I haven't thought of. An attacker would not be affected by latency, since he's chaining his own blocks, so he would have an advantage. The chain would temporarily fork more often due to latency.

The final number I'm interested in is the 500kb limit on block sizes.
According to Wikipedia, Visa alone processed 62 billion transactions
in 2009. Dividing through we get an average of 2000 transactions per
second, so peak rate is probably around double that at 4000
transactions/sec. With a ten minute block target, at peak a block
might need to contain 2.4 million transactions, which just won't fit
into 500kb. Is this 500kb a temporary limitation that will be slowly
removed over time from the official client or something more
fundamental?

A higher limit can be phased in once we have actual use closer to the limit and make sure it's working OK.

Eventually when we have client-only implementations, the block chain size won't matter much. Until then, while all users still have to download the entire block chain to start, it's nice if we can keep it down to a reasonable size.

With very high transaction volume, network nodes would consolidate and there would be more pooled mining and GPU farms, and users would run client-only. With dev work on optimising and parallelising, it can keep scaling up.

Whatever the current capacity of the software is, it automatically grows at the rate of Moore's Law, about 60% per year.

```
diff -u old\db.cpp new\db.cpp
--- old\db.cpp  Sat Dec 18 18:35:59 2010
+++ new\db.cpp  Sun Dec 19 20:53:59 2010
@@ -464,29 +464,32 @@
     ReadBestInvalidWork(bnBestInvalidWork);

     // Verify blocks in the best chain
-    CBlockIndex* pindexFork = NULL;
-    for (CBlockIndex* pindex = pindexBest; pindex && pindex->pprev; pindex = pindex->pprev)
+    if (!fClient)
     {
-        if (pindex->nHeight < nBestHeight-2500 && !mapArgs.count("-checkblocks"))
-            break;
-        CBlock block;
-        if (!block.ReadFromDisk(pindex))
-            return error("LoadBlockIndex() : block.ReadFromDisk failed");
-        if (!block.CheckBlock())
+        CBlockIndex* pindexFork = NULL;
+        for (CBlockIndex* pindex = pindexBest; pindex && pindex->pprev; pindex = pindex->pprev)
         {
```

```
-           printf("LoadBlockIndex() : *** found bad block at %d, hash=%s\n", pindex->nHeight, pindex-
>GetBlockHash().ToString().c_str());
-           pindexFork = pindex->pprev;
+           if (pindex->nHeight < nBestHeight-2500 && !mapArgs.count("-checkblocks"))
+               break;
+           CBlock block;
+           if (!block.ReadFromDisk(pindex))
+               return error("LoadBlockIndex() : block.ReadFromDisk failed");
+           if (!block.CheckBlock())
+           {
+               printf("LoadBlockIndex() : *** found bad block at %d, hash=%s\n", pindex->nHeight,
pindex->GetBlockHash().ToString().c_str());
+               pindexFork = pindex->pprev;
+           }
+       }
+       if (pindexFork)
+       {
+           // Reorg back to the fork
+           printf("LoadBlockIndex() : *** moving best chain pointer back to block %d\n", pindexFork-
>nHeight);
+           CBlock block;
+           if (!block.ReadFromDisk(pindexFork))
+               return error("LoadBlockIndex() : block.ReadFromDisk failed");
+           CTxDB txdb;
+           block.SetBestChain(txdb, pindexFork);
        }
-   }
-   if (pindexFork)
-   {
-       // Reorg back to the fork
-       printf("LoadBlockIndex() : *** moving best chain pointer back to block %d\n", pindexFork-
>nHeight);
-       CBlock block;
-       if (!block.ReadFromDisk(pindexFork))
-           return error("LoadBlockIndex() : block.ReadFromDisk failed");
-       CTxDB txdb;
-       block.SetBestChain(txdb, pindexFork);
    }

    return true;
diff -u old\main.cpp new\main.cpp
--- old\main.cpp        Sat Dec 18 18:35:59 2010
+++ new\main.cpp        Sun Dec 19 20:53:59 2010
@@ -637,6 +637,9 @@
    if (!IsStandard())
        return error("AcceptToMemoryPool() : nonstandard transaction type");

+    if (fClient)
+        return true;
+
    // Do we already have it?
    uint256 hash = GetHash();
    CRITICAL_BLOCK(cs_mapTransactions)
```

```
@@ -1308,23 +1311,26 @@
     if (!CheckBlock())
         return false;

-    //// issue here: it doesn't know the version
-    unsigned int nTxPos = pindex->nBlockPos + ::GetSerializeSize(CBlock(), SER_DISK) - 1 +
GetSizeOfCompactSize(vtx.size());
-
-    map<uint256, CTxIndex> mapUnused;
-    int64 nFees = 0;
-    foreach(CTransaction& tx, vtx)
+    if (!fClient)
     {
-        CDiskTxPos posThisTx(pindex->nFile, pindex->nBlockPos, nTxPos);
-        nTxPos += ::GetSerializeSize(tx, SER_DISK);
+        //// issue here: it doesn't know the version
+        unsigned int nTxPos = pindex->nBlockPos + ::GetSerializeSize(CBlock(), SER_DISK) - 1 +
GetSizeOfCompactSize(vtx.size());
+
+        map<uint256, CTxIndex> mapUnused;
+        int64 nFees = 0;
+        foreach(CTransaction& tx, vtx)
+        {
+            CDiskTxPos posThisTx(pindex->nFile, pindex->nBlockPos, nTxPos);
+            nTxPos += ::GetSerializeSize(tx, SER_DISK);

-        if (!tx.ConnectInputs(txdb, mapUnused, posThisTx, pindex, nFees, true, false))
+            if (!tx.ConnectInputs(txdb, mapUnused, posThisTx, pindex, nFees, true, false))
+                return false;
+        }
+
+        if (vtx[0].GetValueOut() > GetBlockValue(pindex->nHeight, nFees))
             return false;
     }

-    if (vtx[0].GetValueOut() > GetBlockValue(pindex->nHeight, nFees))
-        return false;
-
     // Update block index on disk without changing it in memory.
     // The memory index structure will be changed after the db commits.
     if (pindex->pprev)
@@ -1378,7 +1384,7 @@
     foreach(CBlockIndex* pindex, vDisconnect)
     {
         CBlock block;
-        if (!block.ReadFromDisk(pindex))
+        if (!block.ReadFromDisk(pindex, !fClient))
             return error("Reorganize() : ReadFromDisk for disconnect failed");
         if (!block.DisconnectBlock(txdb, pindex))
             return error("Reorganize() : DisconnectBlock failed");
@@ -1395,7 +1401,7 @@
     {
         CBlockIndex* pindex = vConnect[i];
         CBlock block;
-        if (!block.ReadFromDisk(pindex))
+        if (!block.ReadFromDisk(pindex, !fClient))
             return error("Reorganize() : ReadFromDisk for connect failed");
         if (!block.ConnectBlock(txdb, pindex))
         {
@@ -1526,7 +1532,7 @@

     txdb.Close();

-    if (pindexNew == pindexBest)
+    if (!fClient && pindexNew == pindexBest)
     {
         // Notify UI to display prev block's coinbase if it was ours
         static uint256 hashPrevBestCoinBase;
@@ -1547,10 +1553,6 @@
     // These are checks that are independent of context
     // that can be verified before saving an orphan block.

-    // Size limits
-    if (vtx.empty() || vtx.size() > MAX_BLOCK_SIZE || ::GetSerializeSize(*this, SER_NETWORK) >
MAX_BLOCK_SIZE)
-        return error("CheckBlock() : size limits failed");
-
     // Check proof of work matches claimed amount
     if (!CheckProofOfWork(GetHash(), nBits))
         return error("CheckBlock() : proof of work failed");
@@ -1559,6 +1561,13 @@
     if (GetBlockTime() > GetAdjustedTime() + 2 * 60 * 60)
         return error("CheckBlock() : block timestamp too far in the future");

+    if (fClient && vtx.empty())
+        return true;
+
+    // Size limits
+    if (vtx.empty() || vtx.size() > MAX_BLOCK_SIZE || ::GetSerializeSize(*this, SER_NETWORK) >
MAX_BLOCK_SIZE)
+        return error("CheckBlock() : size limits failed");
+
     // First transaction must be coinbase, the rest must not be
     if (vtx.empty() || !vtx[0].IsCoinBase())
         return error("CheckBlock() : first tx is not coinbase");
@@ -1623,13 +1632,14 @@
         return error("AcceptBlock() : out of disk space");
     unsigned int nFile = -1;
     unsigned int nBlockPos = 0;
-    if (!WriteToDisk(nFile, nBlockPos))
-        return error("AcceptBlock() : WriteToDisk failed");
+    if (!fClient)
+        if (!WriteToDisk(nFile, nBlockPos))
+            return error("AcceptBlock() : WriteToDisk failed");
     if (!AddToBlockIndex(nFile, nBlockPos))
         return error("AcceptBlock() : AddToBlockIndex failed");
     // Relay inventory, but don't relay old inventory during initial block download
-    if (hashBestChain == hash)
+    if (!fClient && hashBestChain == hash)
         CRITICAL_BLOCK(cs_vNodes)
             foreach(CNode* pnode, vNodes)
                 if (nBestHeight > (pnode->nStartingHeight != -1 ? pnode->nStartingHeight - 2000 :
55000))
@@ -2405,6 +2415,8 @@
         {
             if (fShutdown)
                 return true;
+            if (fClient && inv.type == MSG_TX)
+                continue;
             pfrom->AddInventoryKnown(inv);

             bool fAlreadyHave = AlreadyHave(txdb, inv);
@@ -2441,6 +2453,9 @@

             if (inv.type == MSG_BLOCK)
             {
+                if (fClient)
+                    return true;
+
                 // Send block from disk
                 map<uint256, CBlockIndex*>::iterator mi = mapBlockIndex.find(inv.hash);
                 if (mi != mapBlockIndex.end())
@@ -2486,6 +2501,8 @@

     else if (strCommand == "getblocks")
     {
+        if (fClient)
+            return true;
         CBlockLocator locator;
         uint256 hashStop;
         vRecv >> locator >> hashStop;
@@ -2556,6 +2573,8 @@

     else if (strCommand == "tx")
     {
+        if (fClient)
+            return true;
         vector<uint256> vWorkQueue;
         CDataStream vMsg(vRecv);
         CTransaction tx;
@@ -2620,6 +2639,33 @@

         if (ProcessBlock(pfrom, &block))
             mapAlreadyAskedFor.erase(inv);
+    }
+
+
+    else if (strCommand == "headers")
+    {
+        if (!fClient)
+            return true;
+        vector<CBlock> vHeaders;
+        vRecv >> vHeaders;
+
+        uint256 hashBestBefore = hashBestChain;
+        foreach(CBlock& block, vHeaders)
+        {
+            block.vtx.clear();
+
+            printf("received header %s\n", block.GetHash().ToString().substr(0,20).c_str());
+
+            CInv inv(MSG_BLOCK, block.GetHash());
+            pfrom->AddInventoryKnown(inv);
+
+            if (ProcessBlock(pfrom, &block))
+                mapAlreadyAskedFor.erase(inv);
+        }
+
+        // Request next batch
+        if (hashBestChain != hashBestBefore)
+            pfrom->PushGetBlocks(pindexBest, uint256(0));
     }


diff -u old\main.h new\main.h
--- old\main.h  Sat Dec 18 18:35:59 2010
+++ new\main.h  Sun Dec 19 20:53:59 2010
@@ -619,6 +619,8 @@

     bool ReadFromDisk(CDiskTxPos pos, FILE** pfileRet=NULL)
     {
+        assert(!fClient);
+
         CAutoFile filein = OpenBlockFile(pos.nFile, 0, pfileRet ? "rb+" : "rb");
         if (!filein)
             return error("CTransaction::ReadFromDisk() : OpenBlockFile failed");
@@ -1174,6 +1176,7 @@

     bool ReadFromDisk(unsigned int nFile, unsigned int nBlockPos, bool fReadTransactions=true)
     {
+        assert(!fClient);
         SetNull();

         // Open history file to read
@@ -1231,7 +1234,7 @@


 //
-// The block chain is a tree shaped structure starting with the
+// The block index is a tree shaped structure starting with the
 // genesis block at the root, with each block potentially having multiple
 // candidates to be the next block.  pprev and pnext link a path through the
 // main/longest chain.  A blockindex may have multiple pprev pointing back
diff -u old\net.cpp new\net.cpp
--- old\net.cpp Wed Dec 15 22:33:09 2010
+++ new\net.cpp Sun Dec 19 21:51:27 2010
@@ -51,7 +51,15 @@
     pindexLastGetBlocksBegin = pindexBegin;
     hashLastGetBlocksEnd = hashEnd;

-    PushMessage("getblocks", CBlockLocator(pindexBegin), hashEnd);
+    /// Client todo: After the initial block header download, start using getblocks
+    /// here instead of getheaders.  For blocks generated after the first time the
+    /// program was run, we need to download full blocks to watch for received
+    /// transactions in them.  We're able to download headers only for blocks
+    /// generated before we ever ran because they can't contain txes for us.
+    if (::fClient)
+        PushMessage("getheaders", CBlockLocator(pindexBegin), hashEnd);
+    else
+        PushMessage("getblocks", CBlockLocator(pindexBegin), hashEnd);
 }
```

From: **Mike Hearn** <mike@plan99.net>
Date: Thu, Dec 30, 2010 at 12:27 AM
To: Satoshi Nakamoto <satoshin@gmx.com>

Thanks for the info.

I reached the same conclusions about client only nodes and this is what I've been implementing. I'm nearly there ..... I have block chain download, parsing and verification of the blocks/transactions done, with creation of spend transactions almost done.

v1 will basically do as you propose, with the possible optimization of storing only the blocks needed to form the block locator (with the exponential thinning). As Android provides local storage that is private to the app, you don't need to store the entire block chain to be able to accept new blocks ... just enough to ensure you can always stay on the longest chain.

By the way, your code is easy to read and has been an invaluable reference. So thanks for that.

In v2 I'm thinking of showing transactions before they are integrated into the block chain by running secure/locked down relay nodes that send messages to the phones when a transaction is accepted into the memory pool. Android provides a secure, low power back channel to every phone. Messages are stored server side if the device is offline and apps are automatically started on the phone to handle incoming messages.

So as long as the relay nodes are unhacked, this system should give
enough trust that low value transactions can be shown in the UI
immediately. It introduces some centralization/single points of
failure, but if the relay mechanism dies or is hacked, the damage only
lasts for 10 minutes until the new blocks are downloaded.

> Client-only re-implementations would not need to implement EvalScript at
> all, or at most just implement the five ops used by the standard transaction
> templates."

Indeed, there's no point in client-only implementations implementing
EvalScript because they can't verify transactions aren't being double
spent without storing and indexing the entire block chain. My code
parses the scripts and then relies on them having a standard
structure, but doesn't actually run them.

> Educated guess, and the maths work out to round numbers. I wanted something
> that would be not too low if it was very popular and not too high if it
> wasn't.

It'd be interesting to see the working for this. In some sense the
number of coins is arbitrary as the nanocoin representation means the
issuance is so huge it's practically infinite.

> A higher limit can be phased in once we have actual use closer to the limit
> and make sure it's working OK.

It'd be worth implementing some kind of more robust auto update
mechanism, or a schedule for the phase in of this, if only because
when people evaluate "is BitCoin worth my time and effort" a solid
plan for scaling up is good to have written down.

I'm not worried about the physical capabilities of the hardware, but
more protocol ossification as the app is reimplemented and nodes which
don't auto-update themselves increase in number. Client only
reimplementations pose no problems of course, but other systems like
SMTP have proven impossible to globally upgrade despite having
extension mechanisms built in .... just too many implementations and
too many installations.

From: **Satoshi Nakamoto** <satoshin@gmx.com>
Date: Fri, Jan 7, 2011 at 1:00 PM
To: Mike Hearn <mike@plan99.net>

I reached the same conclusions about client only nodes and this is
what I've been implementing. I'm nearly there ..... I have block chain
download, parsing and verification of the blocks/transactions done,
with creation of spend transactions almost done.

That's great! The first client-only implementation will really start to move things to the next step. Is it going to be open source, or Google proprietary?

From: **Mike Hearn** <mike@plan99.net>
Date: Fri, Jan 7, 2011 at 1:24 PM
To: Satoshi Nakamoto <satoshin@gmx.com>

> That's great! The first client-only implementation will really start to
> move things to the next step. Is it going to be open source, or Google
> proprietary?

Open source. It has to be - I am developing it as a personal project
in my spare time and Googles policy is that this is only allowed if
you open source the results. But I would have done that anyway.

I managed to spend my first coins on the testnet with my app a few
days ago, hopefully will get another chance to make progress this
weekend. Probably will have something to show publically sometime in
Feb, touch wood.

From: **Satoshi Nakamoto** <satoshin@gmx.com>
Date: Mon, Jan 10, 2011 at 4:34 PM
To: Mike Hearn <mike@plan99.net>

Open source.

Perfect. Once your code shows how to simplify it down, other authors can follow your lead. Client is a less daunting challenge than full implementation. If it's within reach of more developers, they'll come up with more polished UI and other things I didn't think of. I expect the original software will become the industrial old thing used by GPU farms and pool servers.

BTW, later a good feature for a client version is to keep your private keys encrypted and you give your password each time you send.

I managed to spend my first coins on the testnet with my app a few
days ago, hopefully will get another chance to make progress this
weekend. Probably will have something to show publically sometime in
Feb, touch wood.

Great, keep me updated.

I wanted something
that would be not too low if it was very popular and not too high if it
wasn't.

It'd be interesting to see the working for this. In some sense the
number of coins is arbitrary as the nanocoin representation means the
issuance is so huge it's practically infinite.

It works out to an even 10 minutes per block:
21000000 / (50 BTC * 24hrs * 365days * 4years * 2) = 5.99 blocks/hour

I fudged it to 364.58333 days/year. The halving of 50 BTC to 25 BTC is after 210000 blocks or around 3.9954 years, which is approximate anyway based on the retargeting mechanism's best effort.

I thought about 100 BTC and 42 million, but 42 million seemed high.

I wanted typical amounts to be in a familiar range. If you're tossing around 100000 units, it doesn't feel scarce. The brain is better able to work with numbers from 0.01 to 1000.

If it gets really big, the decimal can move two places and cents become the new coins.

From: **Mike Hearn** <mike@plan99.net>
Date: Mon, Jan 10, 2011 at 4:48 PM
To: Satoshi Nakamoto <satoshin@gmx.com>

Ah, of course, that makes sense.

By the way, if you didn't see it already, there's a discussion on the security of secp256k1 on the forum:

http://www.bitcoin.org/smf/index.php?topic=2699.0

Hal (i presume this is Hal Finney) seems to think the curve is at higher risk of attack than random curves. I guess you chose secp256k1 for the mentioned performance improvement?

From: **Satoshi Nakamoto** <satoshin@gmx.com>
Date: Mon, Jan 10, 2011 at 8:47 PM
To: Mike Hearn <mike@plan99.net>

By the way, if you didn't see it already, there's a discussion on the security of secp256k1 on the forum:

http://www.bitcoin.org/smf/index.php?topic=2699.0

Hal (i presume this is Hal Finney)

Yes, it's him. He was supportive on the Cryptography list and ran one of the first nodes.

seems to think the curve is at higher risk of attack than random curves. I guess you chose secp256k1 for the mentioned performance improvement?

I must admit, this project was 2 years of development before release, and I could only spend so much time on each of the many issues. I found guidance on the recommended size for SHA and RSA, but nothing on ECDSA which was relatively new. I took the recommended key size for RSA and converted to equivalent key size for ECDSA, but then increased it so the whole app could be said to be 256-bit security. I didn't find anything to recommend a curve type so I just... picked one. Hopefully there is enough key size to make up for any deficiency.

At the time, I was concerned whether the bandwidth and storage sizes would be practical even with ECDSA. RSA's huge keys were out of the question. Storage and bandwidth seemed tighter back then. I felt the size was either only just becoming practical, or would be soon. When I presented it, I was surprised nobody else was concerned about size, though I was also surprised how many issues they argued, and more surprised that every single one was something I had thought of and solved.

As it turns out, ECDSA verification time may be the greater bottleneck. (In my tests, OpenSSL was taking 3.5ms per ECDSA verify, or about 285 verifies per second) Client versions bypass the problem.

As things have evolved, the number of people who need to run full nodes is less than I originally imagined. The network would be fine with a small number of nodes if processing load becomes heavy.