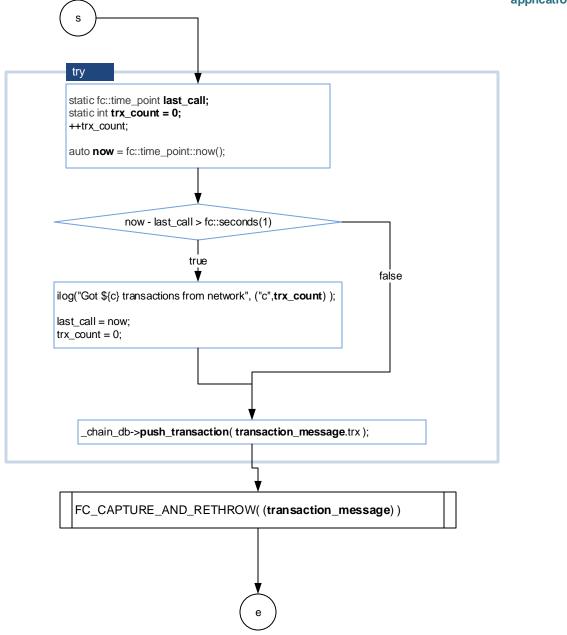


bool application_impl::handle_block(const graphene::net::block_message& blk_msg, bool sync_mode, std::vector<fc::uint160_t>& contained_transaction_message_ids) (17-1)application.cpp @brief allows the application to validate an item prior to broadcasting to peers. @param sync_mode true if the message was fetched through the sync process, false during normal operation @returns true if this message caused the blockchain to switch forks, false if it did not @throws exception if error validating the item, otherwise the item is safe to broadcast on. try auto **latency** = fc::time_point::now() - **blk_msg.**block.timestamp; !sync_mode || blk_msg.block.block_num() % 10000 == 0 true const auto & witness = blk_msg.block.witness(*_chain_db); const auto& witness_account = witness.witness_account(*_chain_db); auto last_irr = _chain_db->get_dynamic_global_properties().last_irreversible_block_num; ilog("Got block: #\${n} \${bid} time: \${t} latency: \${I} ms from: \${w} irreversible: \${i} (-\${d})", ("t",blk_msg.block.timestamp) ("n", **blk_msg.**block.block_num()) ("bid", blk_msg.block.id()) ("I", (latency.count()/1000)) ("w",witness_account.name) ("i",last_irr) ("d",blk_msg.block.block_num()-last_irr)); false FC_ASSERT((latency.count()/1000) > -5000, "Rejecting block with timestamp in the future"); try // TODO: in the case where this block is valid but on a fork that's too old for us to switch to, // you can help the network code out by throwing a block_older_than_undo_history exception. // when the net code sees that, it will stop trying to push blocks from that chain, but // leave that peer connected so that they can get sync blocks from us bool result = _chain_db->push_block(blk_msg.block, (_is_block_producer | _force_validate) ? database::skip_nothing : database::skip_transaction_signatures);

// the block was accepted, so we now know all of the transactions contained in the block

bool application_impl::handle_block(const graphene::net::block_message& blk_msg, bool sync_mode, std::vector<fc::uint160_t>& contained_transaction_message_ids) (17-2)application.cpp // the block was accepted, so we now know all of the transactions contained in the block !sync_mode / if we're not in sync mode, there's a chance we will be seeing some transactions // included in blocks before we see the free-floating transaction itself. If that // happens, there's no reason to fetch the transactions, so construct a list of the // transaction message ids we no longer need. # during sync, it is unlikely that we'll see any old for const processed transaction & transaction: blk_msg.block.transactions true graphene::net::trx_message transaction_message(transaction); false contained_transaction_message_ids.push_back(graphene::net::message(transaction_message).id()); return result; catch (const graphene::chain::unlinkable_block_exception& e) // translate to a graphene::net exception elog("Error when pushing block:\n\${e}", ("e", e.to_detail_string())); FC_THROW_EXCEPTION(graphene::net::unlinkable_block_exception, "Error when pushing block:\n\${e}", ("e", e.to_detail_string())); catch(const fc::exception& e) elog("Error when pushing block:\n\${e}", ("e", e.to_detail_string())); !_is_finished_syncing && !sync_mode throw; true _is_finished_syncing = true; _self->syncing_finished(); false FC_CAPTURE_AND_RETHROW((blk_msg)(sync_mode)) return false;

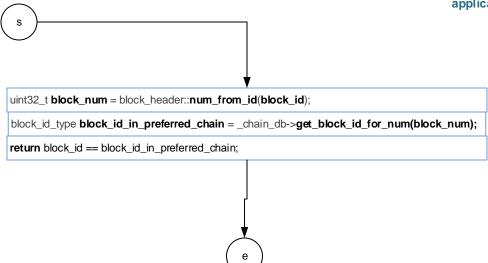
18-1



bool application_impl::is_included_block(const block_id_type& block_id)

application.cpp

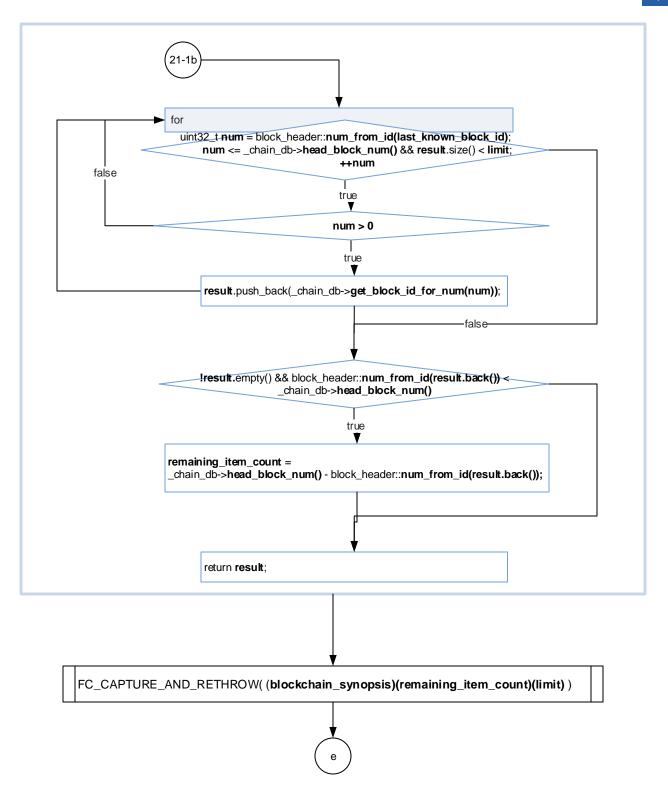
20-1



std::vector<item_hash_t> application_impl::get_block_ids(const std::vector<item_hash_t>& blockchain_synopsis, uint32_t& remaining_item_count, uint32_t limit) application.cpp (21-1)* Assuming all data elements are ordered in some way, this method should start * return up to limit ids that occur *after* the last ID in synopsis that * we recognize. * On return, remaining_item_count will be set to the number of items * in our blockchain after the last item returned in the result, try * or 0 if the result contains the last item in the blockchain vector<block_id_type> result; remaining_item_count = 0; chain db->head block num() == 0 true return result; false result.reserve(limit); block_id_type last_known_block_id; blockchain_synopsis.empty() || (blockchain_synopsis.size() == 1 && blockchain_synopsis[0] == block_id_type()) // peer has sent us an empty synopsis meaning they have no blocks. // A bug in old versions would cause them to send a synopsis containing block 000000000 // when they had an empty blockchain, so pretend they sent the right thing here. false // do nothing, leave last_known_block_id set to zero bool found_a_block_in_synopsis = false; for constitem_hash_t& block_id_in_synopsis boost::adaptors::reverse(blockchain_synopsis) false true block_id_in_synopsis == block_id_type() || (_chain_db->is_known_block(block_id_in_synopsis) && is_included_block(block_id_in_synopsis)) true true last_known_block_id = block_id_in_synopsis; found_a_block_in_synopsis = true; break; -false !found_a_block_in_synopsis FC_THROW_EXCEPTION(graphene::net::peer_is_on_an_unreachable_fork, "Unable to provide a list of blocks starting at any of the blocks in peer's synopsis"); false

application.cpp

(21-2)



std::vector<item_hash_t> application_impl::get_blockchain_synopsis(const item_hash_t& reference_point, uint32_t number_of_blocks_after_reference_point)

application.cpp

```
* Returns a synopsis of the blockchain used for syncing. This consists of a list of
* block hashes at intervals exponentially increasing towards the genesis block.
* When syncing to a peer, the peer uses this data to determine if we're on the same
* fork as they are, and if not, what blocks they need to send us to get us on their
* fork.
* In the over-simplified case, this is a straighforward synopsis of our current
* preferred blockchain; when we first connect up to a peer, this is what we will be sending.
* It looks like this:
* If the blockchain is empty, it will return the empty list.
  If the blockchain has one block, it will return a list containing just that block.
  If it contains more than one block:
   the first element in the list will be the hash of the highest numbered block that
      we cannot undo
   the second element will be the hash of an item at the half way point in the undoable
     segment of the blockchain
   the third will be ~3/4 of the way through the undoable segment of the block chain
   the fourth will be at ~7/8...
   the last item in the list will be the hash of the most recent block on our preferred chain
* so if the blockchain had 26 blocks labeled a - z, the synopsis would be:
* the idea being that by sending a small (<30) number of block ids, we can summarize a huge
* blockchain. The block ids are more dense near the end of the chain where because we are
* more likely to be almost in sync when we first connect, and forks are likely to be short.
* If the peer we're syncing with in our example is on a fork that started at block 'v',
* then they will reply to our synopsis with a list of all blocks starting from block 'u',
* the last block they know that we had in common.
* In the real code, there are several complications.
* First, as an optimization, we don't usually send a synopsis of the entire blockchain, we
* send a synopsis of only the segment of the blockchain that we have undo data for. If their
* fork doesn't build off of something in our undo history, we would be unable to switch, so there's
* no reason to fetch the blocks.
* Second, when a peer replies to our initial synopsis and gives us a list of the blocks they think
* we are missing, they only send a chunk of a few thousand blocks at once. After we get those
* block ids, we need to request more blocks by sending another synopsis (we can't just say "send me
* the next 2000 ids" because they may have switched forks themselves and they don't track what
* they've sent us). For faster performance, we want to get a fairly long list of block ids first,
* then start downloading the blocks.
* The peer doesn't handle these follow-up block id requests any different from the initial request;
* it treats the synopsis we send as our blockchain and bases its response entirely off that. So to
* get the response we want (the next chunk of block ids following the last one they sent us, or,
* failing that, the shortest fork off of the last list of block ids they sent), we need to construct
* a synopsis as if our blockchain was made up of:
* 1. the blocks in our block chain up to the fork point (if there is a fork) or the head block (if no fork)
   2. the blocks we've already pushed from their fork (if there's a fork)
   3. the block ids they've previously sent us
* Segment 3 is handled in the p2p code, it just tells us the number of blocks it has (in
* number_of_blocks_after_reference_point) so we can leave space in the synopsis for them.
* We're responsible for constructing the synopsis of Segments 1 and 2 from our active blockchain and
^{\star} fork database. The reference_point parameter is the last block from that peer that has been
* successfully pushed to the blockchain, so that tells us whether the peer is on a fork or on
* the main chain.
*/
```

std::vector<item_hash_t> application_impl::get_blockchain_synopsis(const item_hash_t& reference_point, uint32 t number of blocks after reference point) 24-1 application.cpp std::vector<item hash t> synopsis; synopsis.reserve(30); uint32_t high_block_num; uint32_t non_fork_high_block_num; uint32_t low_block_num = _chain_db->last_non_undoable_block_num(); std::vector<block_id_type> fork_history; reference_point != item_hash_t() false // the node is asking for a summary of the block chain up to a specified true // block, which may or may not be on a fork // for now, assume it's not on a fork is_included_block(reference_point) false // reference_point is a block we know about and is on the main chain true uint32_t reference_point_block_num = block_header::num_from_id(reference_point); assert(reference_point_block_num > 0); high_block_num = reference_point_block_num; non_fork_high_block_num = high_block_num; reference_point_block_num < low_block_num // we're on the same fork (at least as far as reference_point) but we've passed // reference point and could no longer undo that far if we diverged after that // block. This should probably only happen due to a race condition where // the network thread calls this function, and then immediately pushes a bunch of blocks, // then the main thread finally processes this function. // with the current framework, there's not much we can do to tell the network // thread what our current head block is, so we'll just pretend that // our head is actually the reference point. // this *may* enable us to fetch blocks that we're unable to push, but that should // be a rare case (and correctly handled) low_block_num = reference_point_block_num; true

std::vector<item_hash_t> application_impl::get_blockchain_synopsis(const item_hash_t& reference_point, uint32_t number_of_blocks_after_reference_point) 24-1 application.cpp 24-1b // block is a block we know about, but it is on a fork try fork_history = _chain_db->get_block_ids_on_fork(reference_point); // returns a vector where the last element is the common ancestor with the preferred chain, // and the first element is the reference point you passed in assert(fork_history.size() >= 2); fork_history.front() != reference_point true false edump((fork_history)(reference_point)); assert(fork_history.front() == reference_point); block_id_type last_non_fork_block = fork_history.back(); fork_history.pop_back(); // remove the common ancestor boost::reverse(fork_history); // if the fork goes all the way back to genesis (does graphene's fork db allow this?) last_non_fork_block == block_id_type() -false true non_fork_high_block_num = 0; non_fork_high_block_num = block_header::num_from_id(last_non_fork_block); high_block_num = non_fork_high_block_num + fork_history.size(); assert(high_block_num == block_header::num_from_id(fork_history.back())); catch (const fc::exception& e) elog("Unable to construct a blockchain synopsis for reference hash \${hash}: \${exception}", ("hash", reference_point)("exception", e)); throw; non_fork_high_block_num < low_block_num wlog("Unable to generate a usable synopsis because the peer we're generating it for forked too long ago " our chains diverge after block #\${non_fork_high_block_num} but only undoable to block #\${low_block_num}", ("low_block_num", low_block_num) ("non_fork_high_block_num", non_fork_high_block_num)); FC_THROW_EXCEPTION(graphene::net::block_older_than_undo_history, "Peer is are on a fork I'm unable to switch to");

std::vector<item_hash_t> application_impl::get_blockchain_synopsis(const item_hash_t& reference_point, uint32_t number_of_blocks_after_reference_point) 24-1 application.cpp 24-1a // no reference point specified, summarize the whole block chain high_block_num = _chain_db->head_block_num(); non_fork_high_block_num = high_block_num; high_block_num == 0 false true return synopsis; // we have no blocks low_block_num == 0 true false // at this point: low_block_num = 1; // low_block_num is the block before the first block we can undo, // non_fork_high_block_num is the block before the fork (if the peer is on a fork, or otherwise it is the same as high_block_num) // high_block_num is the block number of the reference block, or the end of the chain if no reference provided // true_high_block_num is the ending block number after the network code appends any item ids it knows about that we don't uint32_t true_high_block_num = high_block_num + number_of_blocks_after_reference_point; do // for each block in the synopsis, figure out where to pull the block id from. // if it's <= non_fork_high_block_num, we grab it from the main blockchain; // if it's not, we pull it from the fork history low block num <= non fork high block num -false true synopsis.push_back(_chain_db->get_block_id_for_num(low_block_num)); true synopsis.push_back(fork_history[low_block_num - non_fork_high_block_num - 1]); low_block_num += (true_high_block_num - low_block_num + 2) / 2; (low_block_num <= high_block_num); false while //idump((synopsis)); return synopsis; // End of first "try" FC_THROW_EXCEPTION(graphene::net::block_older_than_undo_history, "Peer is are on a fork I'm unable to switch to");