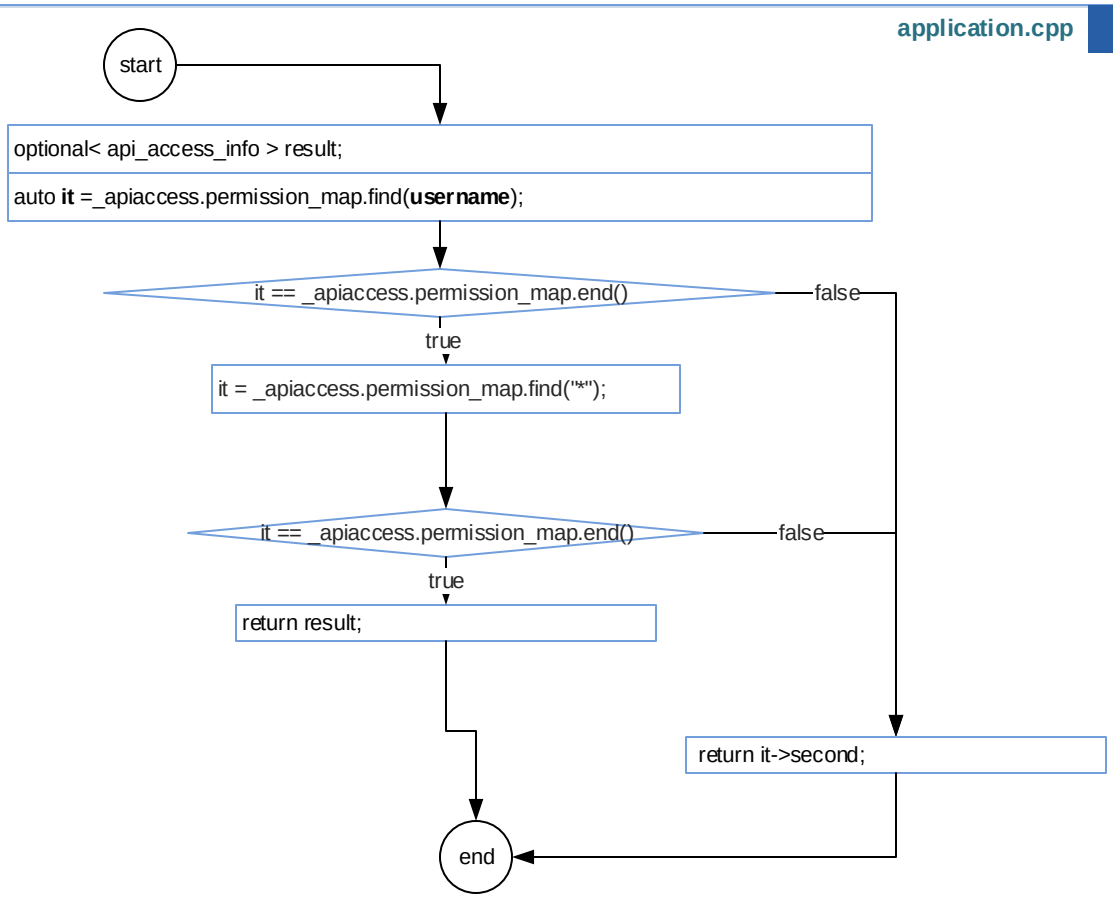
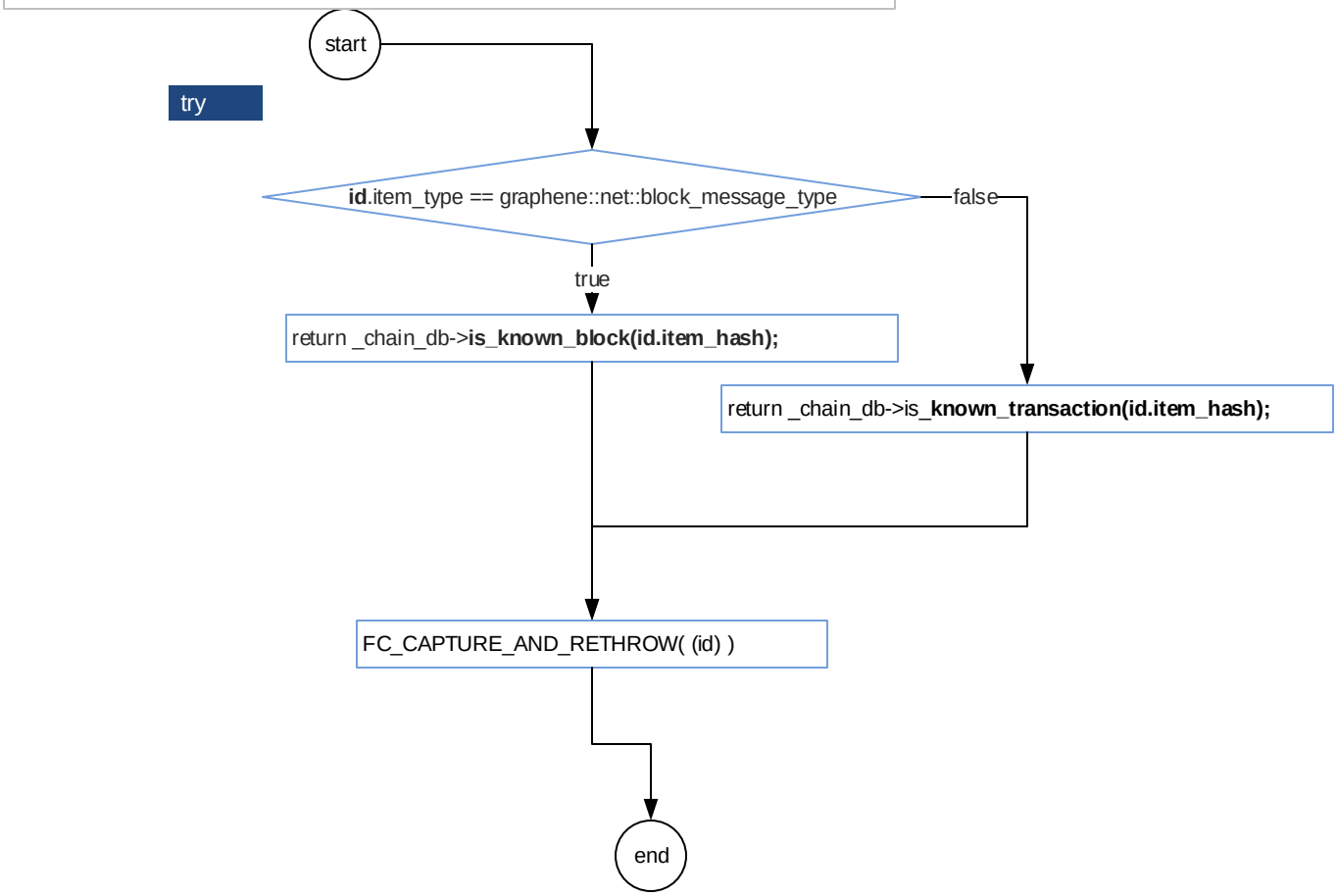

start
optional< api_access_info > result;
auto **it** =_apiaccess.permission_map.find(**username**);
it == _apiaccess.permission_map.end()
false
true
it = _apiaccess.permission_map.find("*");
it == _apiaccess.permission_map.end()
false
true
return result;
return it->second;
end

bool application_impl::**has_item**(const net::item_id& **id**) **1**

application.cpp

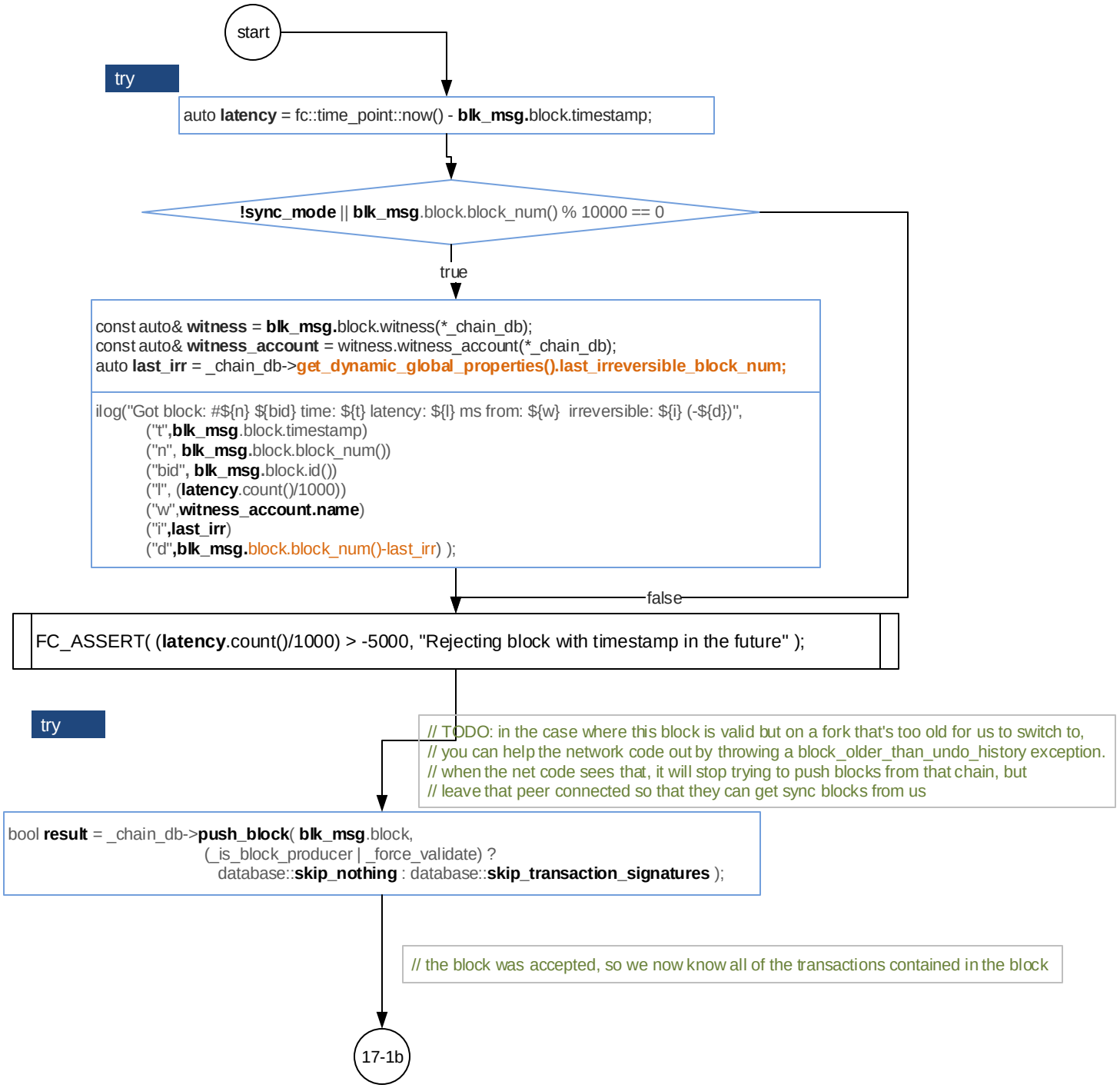
* If delegate has the item, the network has no need to fetch it.



```
bool application_impl::handle_block(const graphene::net::block_message& blk_msg,
                                    bool sync_mode,
                                    std::vector<fc::uint160_t>& contained_transaction_message_ids)
```

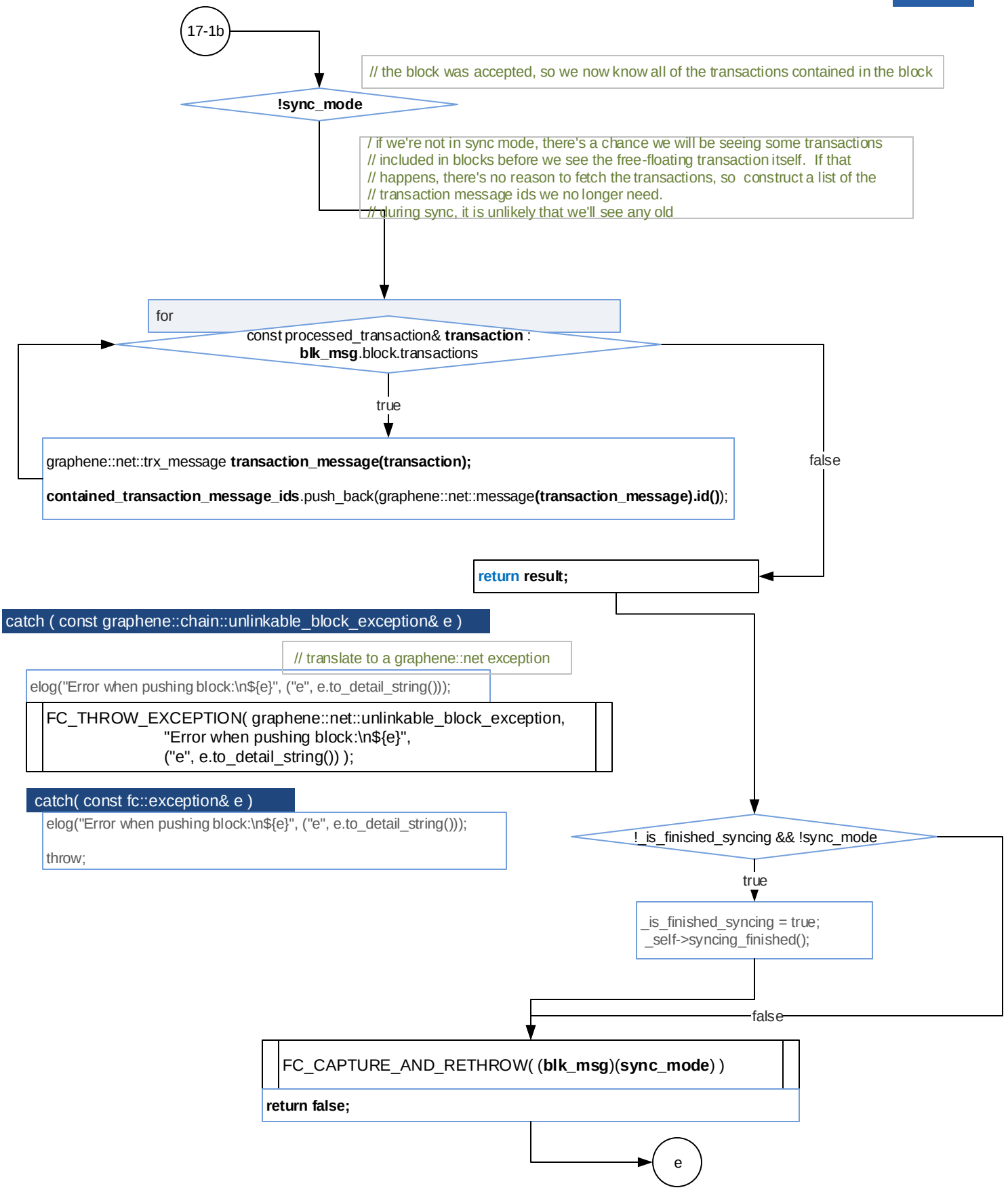
application.cpp (17-1)

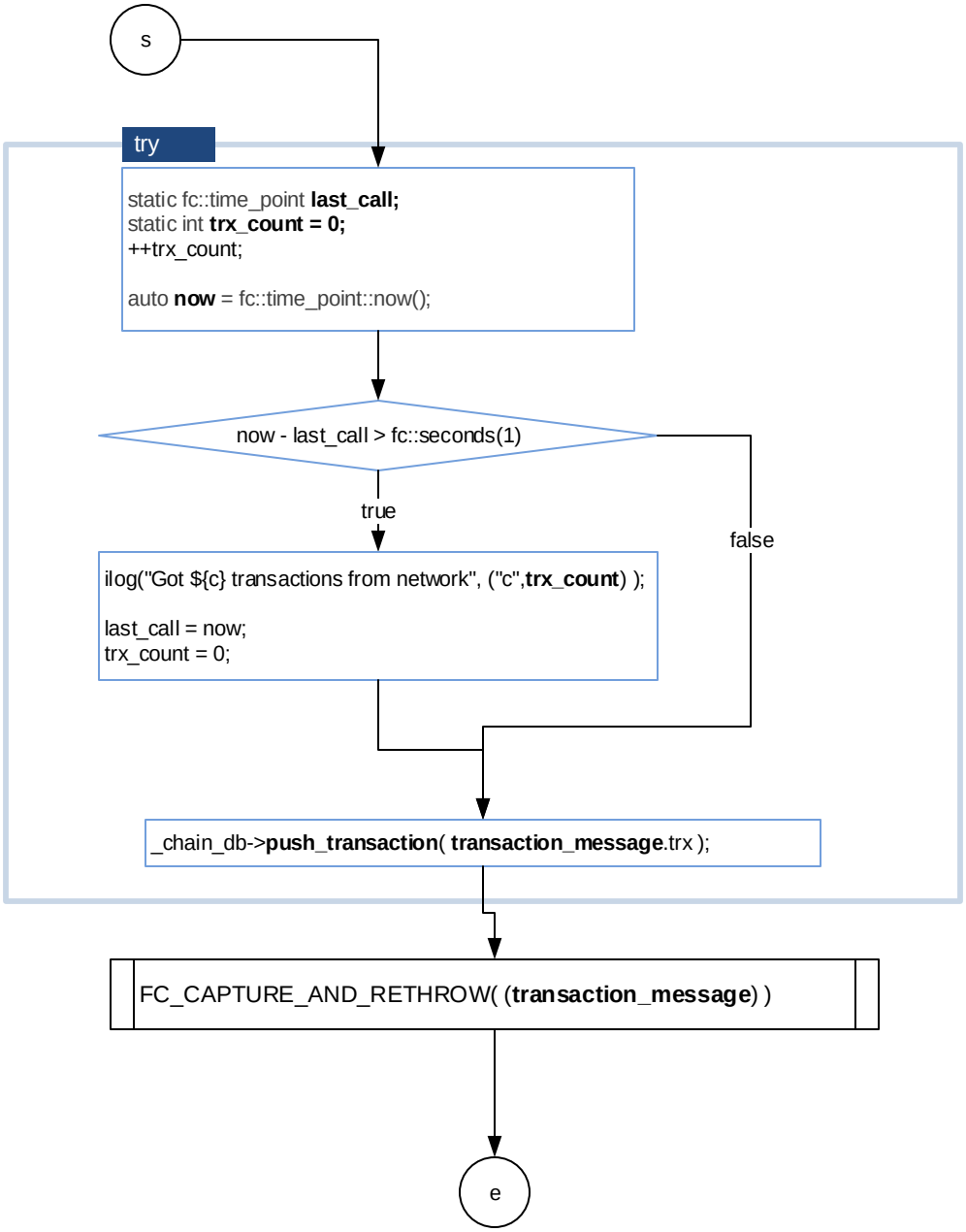
```
@brief allows the application to validate an item prior to broadcasting to peers.

@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.
```



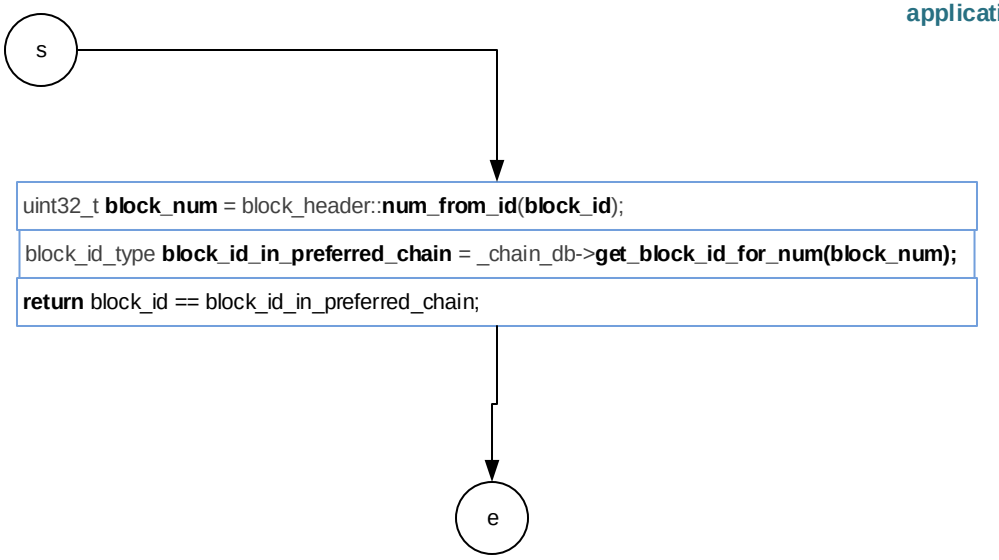
```
bool application_impl::handle_block(const graphene::net::block_message& blk_msg,
                                    bool sync_mode,
                                    std::vector<fc::uint160_t>& contained_transaction_message_ids)
```

application.cpp (17-2)



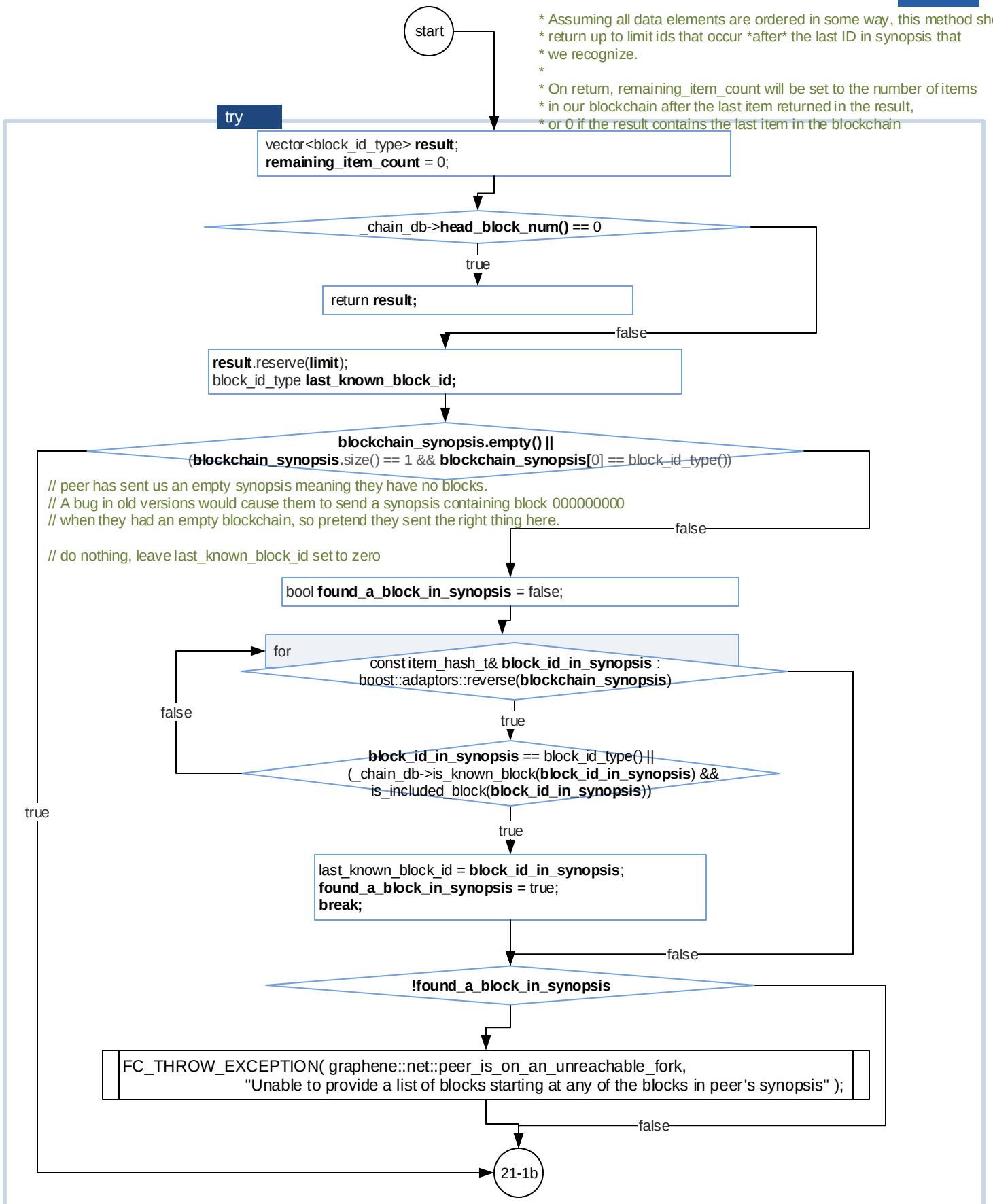

s
try
static fc::time_point last_call;
static int trx_count = 0;
++trx_count;
auto now = fc::time_point::now();
now - last_call > fc::seconds(1)
true
false
ilog("Got ${c} transactions from network", ("c",trx_count));
last_call = now;
trx_count = 0;
_chain_db->push_transaction(transaction_message.trx);
FC_CAPTURE_AND_RETHROW((transaction_message))
e

bool application_impl::**is_included_block**(const block_id_type& **block_id**)

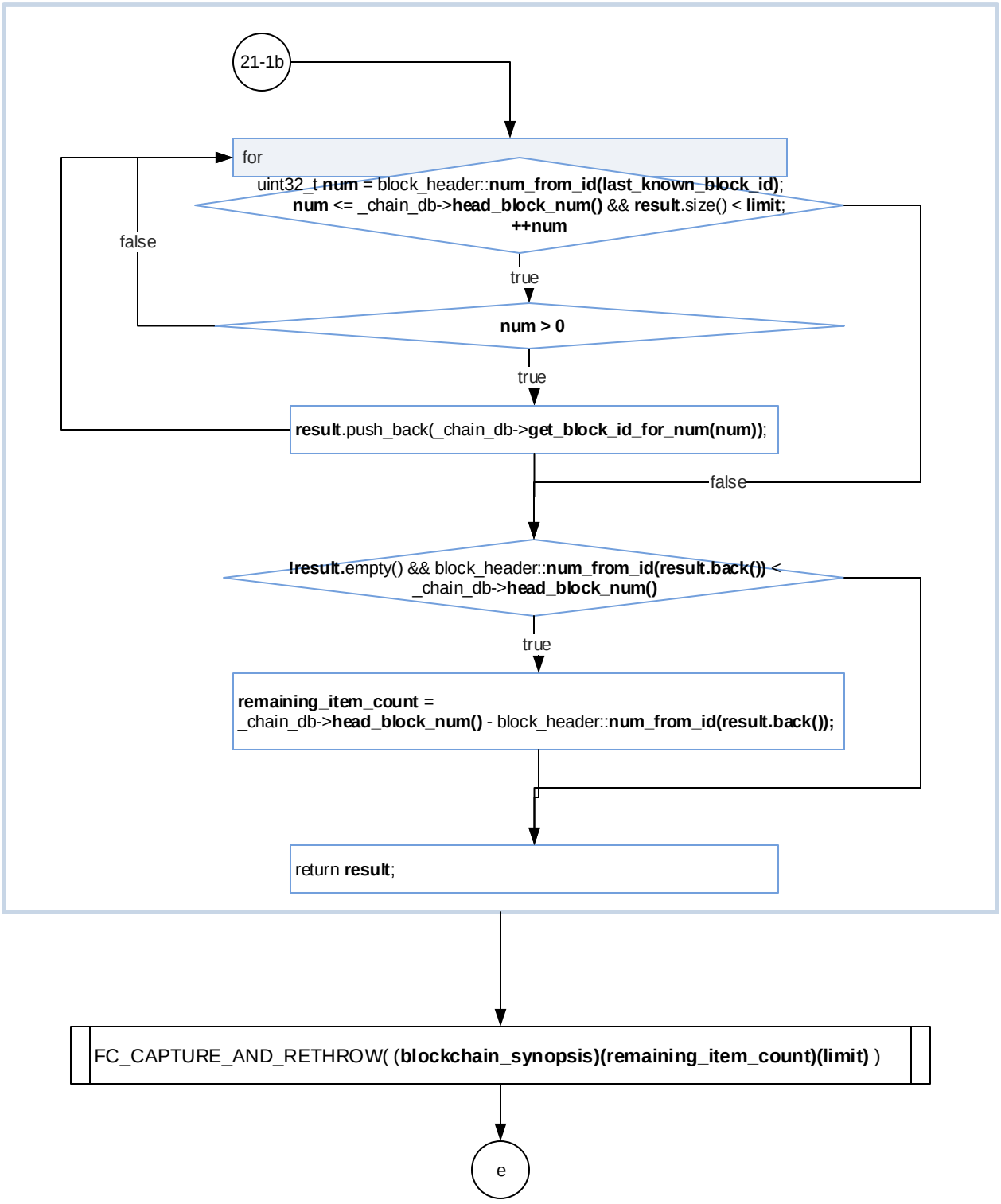


std::vector<item_hash_t> application_impl::**get_block_ids**(const std::vector<item_hash_t>& **blockchain_synopsis**,
uint32_t& **remaining_item_count**,
uint32_t **limit**)

application.cpp (21-1)



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**)



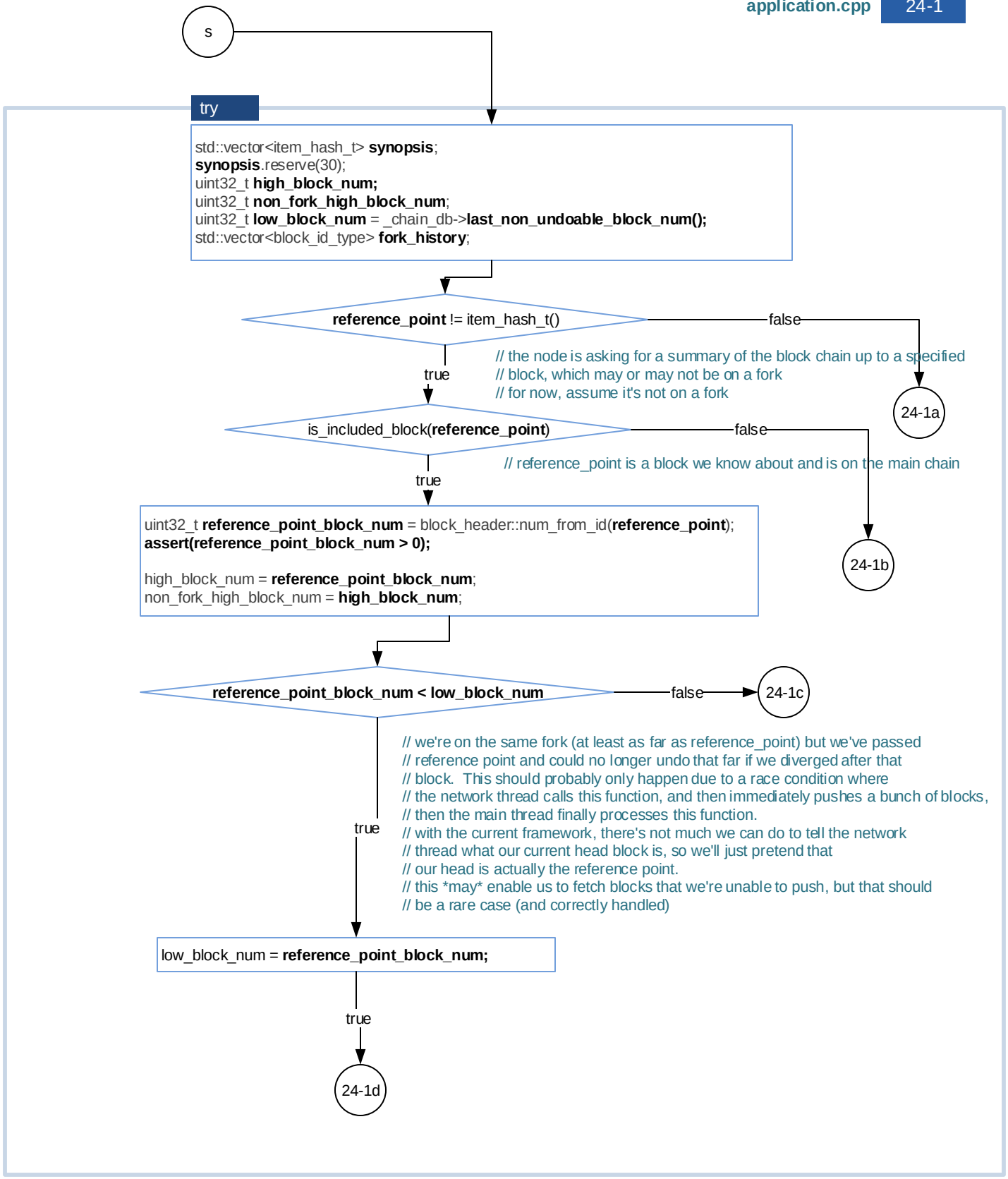
std::vector<item_hash_t> application_impl::**get_blockchain_synopsis**(const item_hash_t& **reference_point,**
uint32_t **number_of_blocks_after_reference_point**)

application.cpp

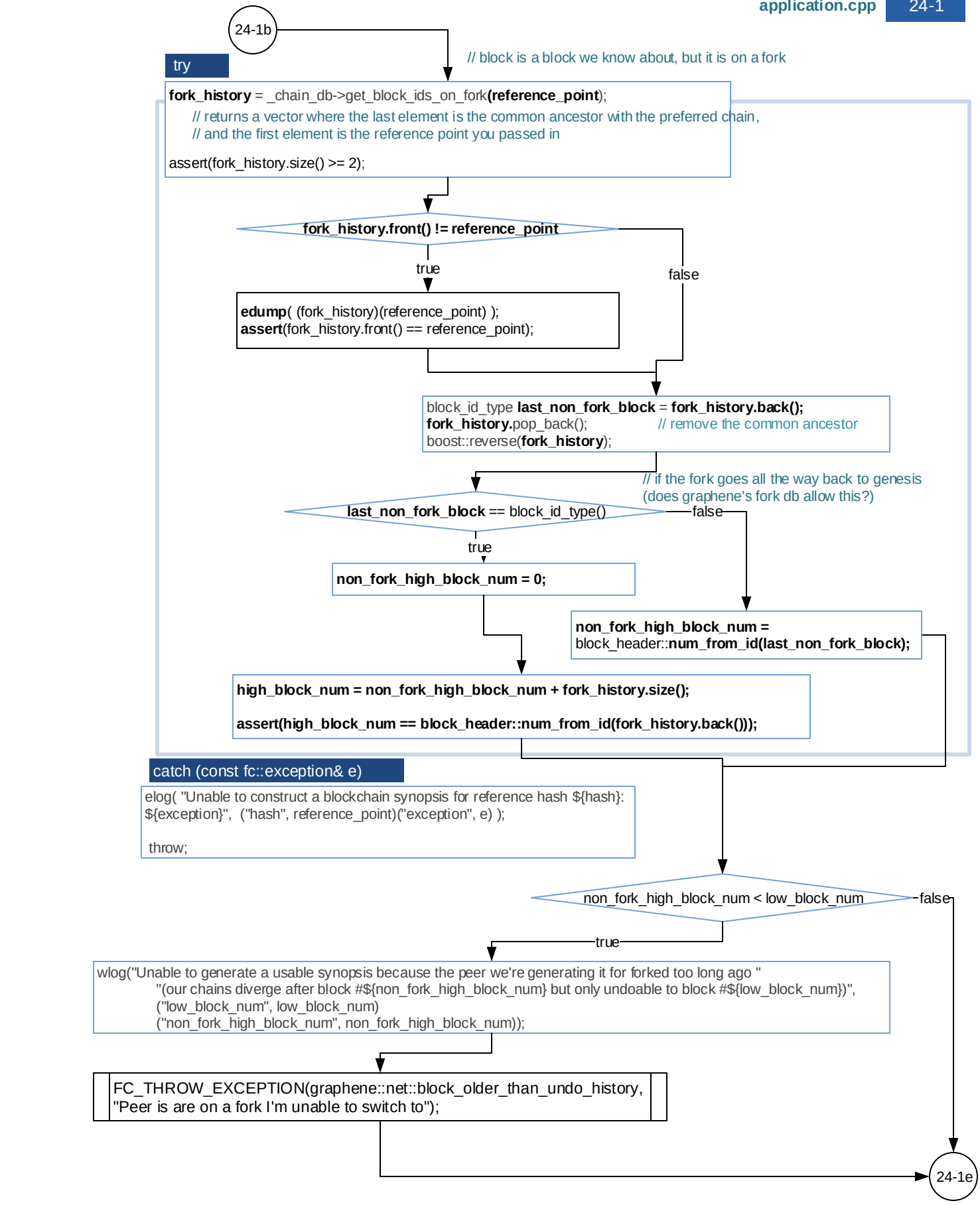
```
/**
 * Returns a synopsis of the blockchain used for syncing.  This consists of a list of
 * block hashes at intervals exponentially increasing towards the genesis block.
 * When syncing to a peer, the peer uses this data to determine if we're on the same
 * fork as they are, and if not, what blocks 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...
 *       &c.
 *     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:
 *    a n u x z
 * 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
 * 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**)

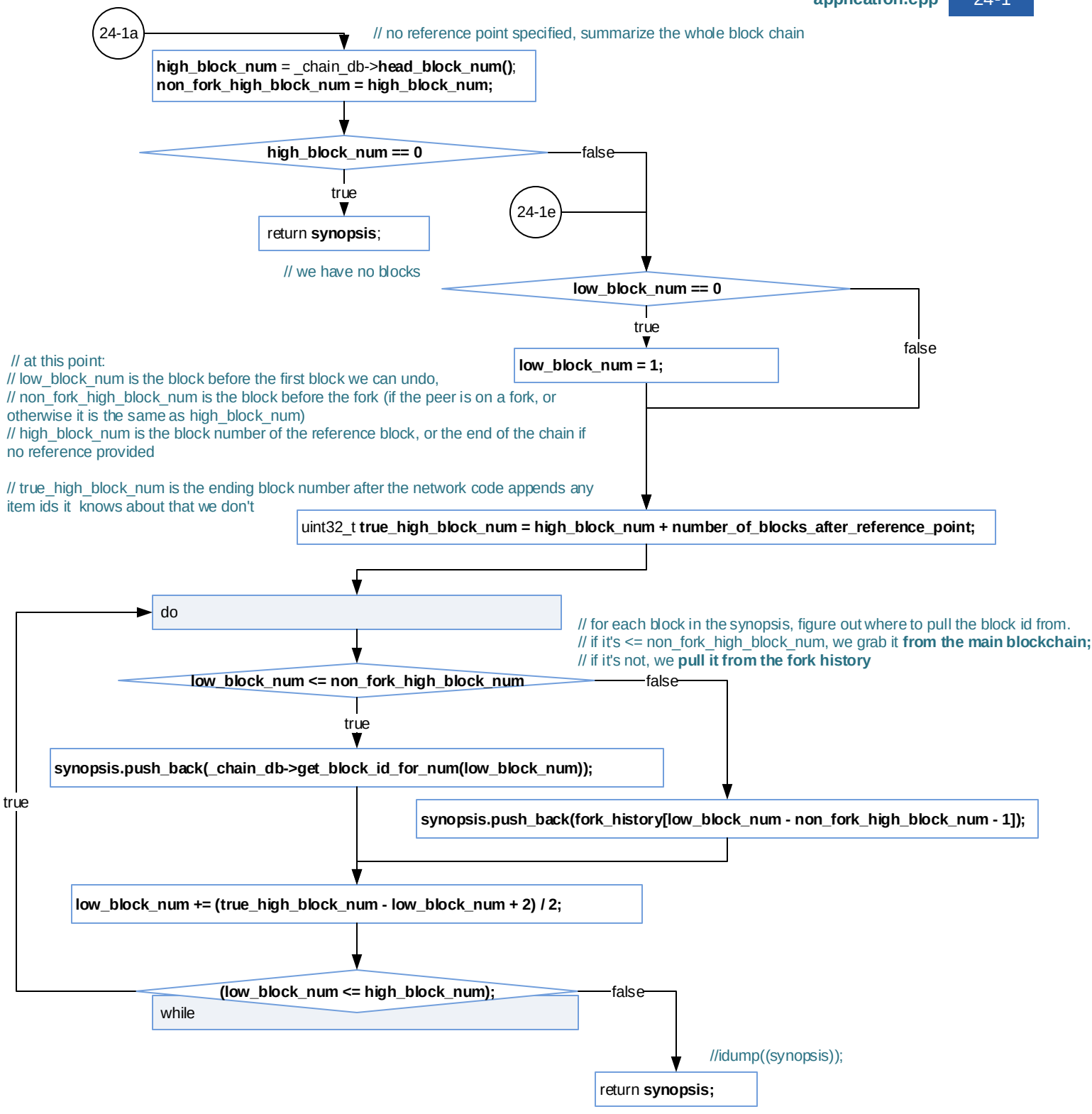
application.cpp 24-1



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



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



// 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");
```