

Swaptacular Messaging Protocol

Evgeni Pandurksi

2023-08-23

Contents

Overview	1
Incoming messages	3
ConfigureAccount	3
PrepareTransfer	6
FinalizeTransfer	8
Outgoing messages	10
RejectedConfig	10
RejectedTransfer	10
PreparedTransfer	11
FinalizedTransfer	13
AccountUpdate	14
AccountPurge	17
AccountTransfer	18
Requirements for Client Implementations	20
RT record	20
Received RejectedTransfer message	21
Received PreparedTransfer message	21
Received FinalizedTransfer message	22
AD record	22
Received AccountUpdate message	22
Received AccountPurge message	23
AL record	23
Received AccountTransfer message	24

Overview

This protocol is centered around two types of actors: *debtors* and *creditors*. A debtor is a person or an organization that manages a digital currency. A creditor is a person or an organization that owns tokens in one or more debtors' digital

currencies. The relationship is asymmetrical: Currency tokens express the fact that the debtor owes something to the creditor. Although a creditor theoretically can have a negative account balance, the relationship is not supposed to work in this direction. The protocol supports the following operations:

1. Creditors can open accounts with debtors.¹
2. Creditors can re-configure existing accounts. Notably, creditors can schedule accounts for deletion, and specify an amount on the account, that is considered negligible.
3. Creditors can safely delete existing accounts with debtors. The emphasis is on *safely*. When the balance on one account is not zero, deleting the account may result in a loss of non-negligible amount of money (tokens of the digital currency). Even if the balance was negligible at the moment of the deletion request, there might have been a pending incoming transfer to the account, which would be lost had the account been deleted without the necessary precautions. To achieve safe deletion, this protocol requires that the account is scheduled for deletion, and the system takes care to delete the account when (and if) it is safe to do so.
4. Creditors can transfer money from their account to other creditors' accounts. Transfers are possible only between account in the same currency (that is: same debtor). The execution of the transfer follows the "two phase commit" paradigm. First the transfer is *prepared*, and then *finalized* (committed or dismissed). A successfully prepared transfer, gives a very high probability for the success of the eventual subsequent *commit*. This paradigm allows many transfers to be committed atomically. Enabling circular exchanges between different currencies is an important goal of this protocol.
5. Creditors receive notification events for every non-negligible transfer in which they participate (that is: all outgoing transfers, and all non-negligible incoming transfers). Those notification events are properly ordered, so that the creditor can reliably assemble the transfer history for each account (the account ledger).
6. Actors other than creditors (called *coordinators*), can make transfers from one creditor's account to another creditor's account. This can be useful for implementing automated direct debit, and a wide range of other automated exchange systems. In fact, when a currency holder (aka creditor) makes a transfer, it is treated as a transfer initiated by a coordinator of type "direct". When a currency issuer (aka debtor) creates new money into existence, this is treated as a transfer initiated by a coordinator of type "issuing".

It is important to note that the currency issuers (aka the debtors) use the same protocol to communicate with the accounting server as the currency holders (aka the creditors). The "only" difference is that issuers' accounts (also called debtors' accounts) will have negative account balances.

¹A given creditor can have *at most one account* with a given debtor. This limitation greatly simplifies the protocol, at the cost of making rare use cases less convenient. (To have more than one account with the same debtor, the creditor will have to use more than one `creditor_ids`.)

The protocol has been designed with the following important properties in mind:

1. In case of prolonged network disconnect, creditors can synchronize their state with the accounting server, without losing data or money.
2. Messages may arrive out-of-order, or be delivered more than once, without causing any problems (with the exception of possible delays).
3. In the case of lost messages, or even a complete database loss on the client's side, eventually, the client should be able to synchronize its state with the accounting server, without losing money (obviously some data may have been lost).
4. The protocol is generic enough to support different "backend" implementations. For example, it should be possible to implement a proxy/adaptor that allows clients that "talk" this protocol to create bank accounts and make bank transfers.
5. The protocol works well both with positive and negative interest rates on creditors' accounts.

This document defines the high-level semantics of the protocol, the mandated behaviors in the protocol, and the structure of the protocol messages (names, types, and descriptions of the fields). This document does not define or mandate any particular method for message serialization and message transport. Those topics will be discussed in separate document(s).

Note: The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Incoming messages

Incoming messages are messages that the clients send to the accounting server. There are 3 types of incoming messages:

ConfigureAccount

Upon receiving this message, the server makes sure that the specified account exists, and updates its configuration settings.²

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with **debtor_id**, identifies the account.³

²As a rough guideline, on average, ConfigureAccount messages for one account should not be sent more often than once per minute.

³All **creditor_ids** between 0 and 4294967295 are reserved. Implementations SHOULD NOT use numbers in this interval for *creditor's accounts*. In particular, implementations SHOULD use the account with **creditor_id** = 0 (*the debtor's account*) to issue new currency tokens in circulation.

negligible_amount : float The maximum amount that can be considered negligible. This MUST be a *finite* non-negative number. It can be used to: 1) decide whether an account can be safely deleted; 2) decide whether an incoming transfer is insignificant; 3) decide whether to allow new currency tokens to be issued (when the account is a debtor's account).

config_flags : int32 Account configuration bit-flags. Different server implementations may use these flags for different purposes.

The lowest 16 bits are reserved. Bit 0 has the meaning "scheduled for deletion". If all of the following conditions are met, an account SHOULD eventually be removed from the server's database:⁴

- The account is "scheduled for deletion".⁵
- At least one day has passed since account's creation.⁶
- Account's configuration have not been updated for at least `MAX_CONFIG_DELAY` seconds.⁷
- There are no outgoing prepared transfers (for which the account is the sender) that await finalization (see `PreparedTransfer`).
- There are no incoming prepared transfers (for which the account is the recipient) that await finalization and have not missed their deadlines already.
- If the account gets removed from the server's database, it is not possible the owner of the account to lose an amount bigger than the **negligible_amount**. Note that unless the negligible amount is huge, or the owner of the account has an alternative way to access his funds, this implies that the account can not receive incoming transfers after being deleted.

If those condition are *not met*, accounts MUST NOT be removed. Some time after an account has been removed from the server's database, an `AccountPurge` message MUST be sent to inform about this.⁸

config_data : string Additional account configuration settings. Different

⁴When an account with a non-zero principal is being deleted, an `AccountTransfer` message SHOULD be sent, informing the owner of the account about the zeroing out of the account's principal before the deletion.

⁵Server implementations must not accept incoming transfers for "scheduled for deletion" accounts. That is: `PrepareTransfer` messages that has a "scheduled for deletion" creditor's account as a recipient MUST be rejected.

⁶Note that an account can be removed from the server's database, and then a new account with the same `debtor_id` and `creditor_id` can be created. In those cases care MUST be taken, so that the newly created account always has a later `creation_date`, compared to the preceding account. The most straightforward way to achieve this is not to remove accounts on the same day on which they have been created.

⁷`MAX_CONFIG_DELAY` determines how far in the past a `ConfigureAccount` message should be in order to be ignored. The intention is to avoid the scenario in which an account is removed from server's database, but an old, wandering `ConfigureAccount` message "resurrects" it.

⁸The `AccountPurge` message delay MUST be long enough to ensure that after clients have received the `AccountPurge` message, if they continue to receive old, wandering `AccountUpdate` messages for the purged account, those messages will be ignored (due to expired `ttl`).

server implementations may use different formats for this field.⁹ An empty string MUST always be a valid value, which represents the default configuration settings.

ts : date-time The moment at which this message was sent (the message's timestamp). For a given account, later ConfigureAccount messages MUST have later or equal timestamps, compared to earlier messages.

seqnum : int32 The sequential number of the message. For a given account, later ConfigureAccount messages SHOULD have bigger sequential numbers, compared to earlier messages. Note that when the maximum **int32** value is reached, the next value SHOULD be **-2147483648** (signed 32-bit integer wrapping).

When server implementations process a ConfigureAccount message, they MUST first verify whether the specified account already exists:

1. If the specified account already exists, the server implementation MUST decide whether the same or a later ConfigureAccount message has been applied already.¹⁰ ¹¹ If the received message turns out to be an old one, it MUST be ignored. Otherwise, an attempt MUST be made to update the account's configuration with the requested new configuration. If the new configuration has been successfully applied, an AccountUpdate message MUST be sent; otherwise a RejectedConfig message MUST be sent.
2. If the specified account does not exist, the message's timestamp MUST be checked. If it is more than **MAX_CONFIG_DELAY** seconds in the past, the message MUST be ignored.¹² Otherwise, an attempt MUST be made to create a new account with the requested configuration settings.¹³ ¹⁴ ¹⁵ If

⁹The UTF-8 encoding of the **config_data** string MUST NOT be longer than 2000 bytes.

¹⁰To decide whether a ConfigureAccount message has been applied already, server implementations MUST compare the values of **ts** and **seqnum** fields in the received message, to the values of these fields in the latest applied ConfigureAccount message. **ts** fields MUST be compared first, and only if they are equal, **seqnum** fields MUST be compared as well.

¹¹Note that when comparing "seqnum" fields, server implementations MUST correctly deal with the possible 32-bit integer wrapping. For example, to decide whether **seqnum2** is later than **seqnum1**, the following expression may be used: $0 < (\text{seqnum2} - \text{seqnum1}) \% 0x100000000 < 0x80000000$. Timestamps must also be compared with care, because precision might have been lost when they were saved to the database.

¹²**MAX_CONFIG_DELAY** determines how far in the past a ConfigureAccount message should be in order to be ignored. The intention is to avoid the scenario in which an account is removed from server's database, but an old, wandering ConfigureAccount message "resurrects" it.

¹³Note that an account can be removed from the server's database, and then a new account with the same **debtor_id** and **creditor_id** can be created. In those cases care MUST be taken, so that the newly created account always has a later **creation_date**, compared to the preceding account. The most straightforward way to achieve this is not to remove accounts on the same day on which they have been created.

¹⁴The principal (the amount that the debtor owes to the creditor, without the interest), and the accumulated interest on newly created accounts MUST be zero.

¹⁵When messages arrive out-of-order, it is possible the server to receive a ConfigureAccount message from a client, which requests a new account to be created with its "scheduled for deletion" flag set. When this happens, server implementations MUST NOT reject to create the account solely for the reason that the "scheduled for deletion" flag is set.

a new account has been successfully created, an AccountUpdate message MUST be sent; otherwise a RejectedConfig message MUST be sent.

PrepareTransfer

Upon receiving this message, the server tries to secure some amount, to eventually make a transfer from sender's account to recipient's account.

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with **debtor_id**, identifies the sender's account.

coordinator_type : string Indicates the subsystem which sent this message. MUST be between 1 and 30 symbols, ASCII only.¹⁶

coordinator_id : int64 Along with **coordinator_type**, identifies the client that sent this message (the *coordinator*).

coordinator_request_id : int64 Along with **coordinator_type** and **coordinator_id**, uniquely identifies this message from the coordinator's point of view, so that the coordinator can pair this request with the received response message.

min_locked_amount : int64 The secured amount MUST be equal or bigger than this value. This value MUST be a non-negative number.¹⁷

max_locked_amount : int64 The secured amount MUST NOT exceed this value. This value MUST be equal or bigger than the value of **min_locked_amount**.

recipient : string A string which (along with **debtor_id**) publicly and globally identifies the recipient's account.¹⁸

min_interest_rate : float Determines the minimal approved interest rate. This instructs the server that if the interest rate on the account becomes lower than this value, the transfer MUST NOT be successful. This can be useful when the transferred amount may need to be decreased if the

¹⁶The coordinator type "direct" is reserved for payments initiated directly by the owner of the account, and for such transfers **coordinator_id** MUST be equal to **creditor_id**; The coordinator type "issuing" is reserved for transfers which create new money into existence, and for such transfers **coordinator_id** MUST be equal to **debtor_id**, and the **creditor_id** of the sender must be 0; "interest" MUST be used for transfers initiated by the interest capitalization service; "delete" MUST be used for transfers which zero out the principal on deleted accounts.

¹⁷If **min_locked_amount** is zero, and there are no other impediments to the transfer, the transfer MUST be prepared successfully even when the amount available on the account is zero or less. (In this case, the secured amount will be zero.) This is useful when the sender wants to verify whether the recipient's account exists and accepts incoming transfers.

¹⁸The account identifier MUST have at most 100 symbols, ASCII only. Different server implementations may use different formats for this identifier. Note that **creditor_id** is an ID which is known only to the client that created the account. The account identifier (along with **debtor_id**), on the other hand, MUST provide enough information to publicly and globally identify the account (an IBAN for example).

interest rate on the account has decreased. The value MUST be *finite* and equal or bigger than -100. Normally, this would be -100.

max_commit_delay : int32 The period (in seconds) during which the prepared transfer can be committed successfully. This instructs the server that the generated **deadline** for the prepared transfer MUST NOT be later than this message's timestamp (the **ts** field) plus **max_commit_delay** seconds. This MUST be a non-negative number. If the client does not want the deadline for the transfer to be shorter than normal, this field should be set to some huge number. Normally, this would be 2147483647.

ts : date-time The moment at which this message was sent (the message's timestamp).

When server implementations process a PrepareTransfer message they:

- MUST NOT allow a transfer without verifying that the recipient's account exists, and does accept incoming transfers.
- MUST NOT allow a transfer in which the sender and the recipient is the same account.
- MUST try to secure *as big amount as possible* within the requested limits (between **min_locked_amount** and **max_locked_amount**).
- MUST guarantee that if a transfer is successfully prepared, the probability for successful commit of the secured amount is very high.¹⁹ ²⁰ Notably, the secured amount MUST be locked, so that until the prepared transfer is finalized, the amount is not available for other transfers.
- If the requested transfer has been successfully prepared, MUST send a PreparedTransfer message, and MUST create a new prepared transfer

¹⁹Note that when the interest rate on a given account is negative, the secured (locked) amount will be gradually consumed by the accumulated interest. Therefore, at the moment of the prepared transfer's commit, it could happen that the committed amount exceeds the remaining amount by a considerable margin. In such cases, the commit should be unsuccessful. Also, note that when a PrepareTransfer request is being processed by the server, it can not be predicted what amount will be available on the sender's account at the time of the transfer's commit. For this reason, when a PreparedTransfer message is sent, the server should set the value of the **demurrage_rate** field correctly, so as to inform the client (the coordinator) about *the worst possible case*.

Here is an example how this may work, from the viewpoint of a coordinator who is trying to commit a conditional transfer: The coordinator sends a PrepareTransfer message for the conditional transfer, which he knows, because of the still unrealized condition, will take up to 1 month to get finalized. Then, a PreparedTransfer message for this transfer is received, with a **locked_amount** of 1000, and a **demurrage_rate** of -79.5 percent. The coordinator figures out that if he keeps this prepared transfer around, and does not finalize it, for each passed month, up to 2% of the locked amount will be eaten up (0.98 to the power of 12 equals 0.795). Therefore, the coordinator can calculate that in order to be certain that, after one month, he will be able to commit this prepared transfer successfully, the committed amount should not exceed 980. (That is: The value of the **committed_amount** field in the FinalizeTransfer message that the coordinator sends to commit the transfer, should not exceed 980.)

²⁰There is a trick that opportunistic creditors may try to evade incurring negative interest on their accounts. The trick is to prepare a transfer from one account to another account for the whole available amount, wait for some long time, then commit the prepared transfer and abandon the first account (which at that point would be significantly in red).

record in the server's database, which stores all the data sent with the `PreparedTransfer` message.

- If the requested transfer can not be prepared, **MUST** send a `RejectedTransfer` message.

An important practical case is when `min_locked_amount` and `max_locked_amount` are both equal to zero. In this case no amount will be secured, and whether the transfer will be successful or not will depend on whether the `committed_amount`, sent with the `FinalizeTransfer` message, will be available at the time of the commit.

FinalizeTransfer

Upon receiving this message, the server finalizes a prepared transfer.

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with `debtor_id`, identifies the sender's account.

transfer_id : int64 The opaque ID generated for the prepared transfer. This ID, along with `debtor_id` and `creditor_id`, uniquely identifies the prepared transfer that has to be finalized.

coordinator_type : string **MUST** contain the value of the `coordinator_type` field in the `PrepareTransfer` message that has been sent to prepare the transfer.

coordinator_id : int64 **MUST** contain the value of the `coordinator_id` field in the `PrepareTransfer` message that has been sent to prepare the transfer.

coordinator_request_id : int64 **MUST** contain the value of the `coordinator_request_id` field in the `PrepareTransfer` message that has been sent to prepare the transfer.

committed_amount : int64 The amount that has to be transferred.²¹ This **MUST** be a non-negative number. A 0 signifies that the transfer **MUST** be dismissed.

transfer_note : string A string that the coordinator (the client that finalizes the prepared transfer) wants the recipient and the sender to see.²²

Server implementations **MAY** further limit on the maximal allowed byte-length of the UTF-8 encoding of this string, as long as the limit is correctly stated in the `transfer_note_max_bytes` field in `AccountUpdate` messages.

If the transfer is being dismissed, this field will be ignored, and **SHOULD** contain an empty string.

²¹The `committed_amount` can be smaller, equal, or bigger than the secured (locked) amount.

²²The UTF-8 encoding of the `transfer_note` string **MUST NOT** be longer than 500 bytes.

transfer_note_format : **string** The format used for the **transfer_note** string. An empty string signifies unstructured text.²³

If the transfer is being dismissed, this field will be ignored, and SHOULD contain an empty string.

ts : **date-time** The moment at which this message was sent (the message's timestamp).

When server implementations process a **FinalizeTransfer** message, they MUST first verify whether a matching prepared transfer exists in server's database:²⁴

1. If the specified prepared transfer exists, server implementations MUST:
 - Try to transfer the **committed_amount** from the sender's account to the recipient's account. (When the committed amount is zero, this would be a no-op.) The transfer SHOULD NOT be allowed if, after the transfer, the *available amount*²⁵ on the sender's account would become negative.²⁶
 - Unlock the remainder of the secured (locked) amount, so that it becomes available for other transfers.
 - Remove the prepared transfer from the server's database.
 - Send a **FinalizedTransfer** message.²⁷ Note that the amount transferred to the recipient's account MUST be either zero (when the transfer has

²³The value of the **transfer_note_format** field MUST match the regular expression `^[0-9A-Za-z.-]{0,8}$`.

²⁴The matching prepared transfer MUST have the same values for the **debtor_id**, **creditor_id**, **transfer_id**, **coordinator_type**, **coordinator_id**, and **coordinator_request_id** fields as the received **FinalizeTransfer** message.

²⁵The *available amount* is the amount that the debtor owes to the creditor (including the accumulated interest), minus the total sum secured (locked) for prepared transfers. Note that the available amount can be a negative number.

²⁶To issue new tokens into existence, the server SHOULD use a special account called "*the debtor's account*" (or "the root account"). The debtor's account is special in the following ways:

- The **creditor_id** for the debtor's account is 0.
- The balance on the debtor's account is allowed to go negative, as long as it does not exceed the configured **negligible_amount** for the account (with a negative sign). This gives debtors agents the option to reliably restrict the total amount that a debtor is allowed to issue.
- Interest is not accumulated on the debtor's account.
- All interest payments to/from creditor's accounts, come from/to the debtor's account.
- **AccountTransfer** messages are not sent for transfers from/to the debtor's account. This eliminates a potentially huge amount of network traffic towards the debtor's account, especially for interest payments.
- Each debtor can use its debtor's account **config_data** text field, to configure various important parameters of the currency (like the interest rate). The format for the **config_data** text field will be specified in separate document(s).
- The debtor's account should always be able to receive incoming transfers, even if it does not exist yet, or is "scheduled for deletion". Transferring money to the debtor's account is equivalent to "destroying" the money.

²⁷If the prepared transfer has been committed successfully, **AccountUpdate** messages will be sent eventually, and for non-negligible transfers, **AccountTransfer** messages will be sent eventually as well.

- been dismissed or unsuccessful), or equal to the `committed_amount` (when the transfer has been successful).
2. If the specified prepared transfer does not exist, the message MUST be ignored.

Outgoing messages

Outgoing messages are messages that the accounting server sends to the clients. There are 7 types of incoming messages:

RejectedConfig

Emitted when a `ConfigureAccount` request has been rejected.

debtor_id : int64 The value of the `debtor_id` field in the rejected message.

creditor_id : int64 The value of the `creditor_id` field in the rejected message.

config_ts : date-time The value of the `ts` field in the rejected message.

config_seqnum : int32 The value of the `seqnum` field in the rejected message.

config_flags : int32 The value of the `config_flags` field in the rejected message.

negligible_amount : float The value of the `negligible_amount` field in the rejected message.

config_data : string The value of the `config_data` field in the rejected message.²⁸

rejection_code : string The reason for the rejection of the `ConfigureAccount` request. MUST be between 0 and 30 symbols, ASCII only.

ts : date-time The moment at which this message was sent (the message's timestamp).

RejectedTransfer

Emitted when a request to prepare a transfer has been rejected.

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with `debtor_id` identifies the sender's account.

coordinator_type : string Indicates the subsystem which requested the transfer. MUST be between 1 and 30 symbols, ASCII only.

²⁸The UTF-8 encoding of the `config_data` string MUST NOT be longer than 2000 bytes.

coordinator_id : int64 Along with **coordinator_type**, identifies the client that requested the transfer (the *coordinator*).

coordinator_request_id : int64 Along with **coordinator_type** and **coordinator_id**, uniquely identifies the rejected request from the coordinator's point of view, so that the coordinator can pair this message with the issued request to prepare a transfer.

status_code : string The reason for the rejection of the transfer. MUST be between 0 and 30 symbols, ASCII only. The value MUST not be "OK".²⁹

total_locked_amount : int64 When the transfer has been rejected due to insufficient available amount, this field SHOULD contain the total sum secured (locked) for prepared transfers on the account. This MUST be a non-negative number.

ts : date-time The moment at which this message was sent (the message's timestamp).

PreparedTransfer

Emitted when a new transfer has been prepared, or to remind that a prepared transfer has to be finalized.

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with **debtor_id** identifies the sender's account.

transfer_id : int64 An opaque ID generated for the prepared transfer. This ID, along with **debtor_id** and **creditor_id**, uniquely identifies the prepared transfer.

coordinator_type : string Indicates the subsystem which requested the transfer. MUST be between 1 and 30 symbols, ASCII only.

coordinator_id : int64 Along with **coordinator_type**, identifies the client that requested the transfer (the *coordinator*).

coordinator_request_id : int64 Along with **coordinator_type** and **coordinator_id**, uniquely identifies the accepted request from the

²⁹The mandatory status codes which MUST be used are:

- "SENDER_IS_UNREACHABLE" signifies that the sender's account does not exist, or can not make outgoing transfers.
- "RECIPIENT_IS_UNREACHABLE" signifies that the recipient's account does not exist, or does not accept incoming transfers.
- "TERMINATED" or anything that starts with "TERMINATED", signifies that the transfer has been terminated due to expired deadline or unapproved interest rate change.
- "TRANSFER_NOTE_IS_TOO_LONG" signifies that the transfer has been rejected because the transfer note's byte-length is too big.
- "INSUFFICIENT_AVAILABLE_AMOUNT" signifies that the transfer has been rejected due to insufficient amount available on the account.

coordinator's point of view, so that the coordinator can pair this message with the issued request to prepare a transfer.

locked_amount : int64 The secured (locked) amount for the transfer. This MUST be a non-negative number.

recipient : string The value of the **recipient** field in the corresponding PrepareTransfer message.

prepared_at : date-time The moment at which the transfer was prepared.

demurrage_rate : float The annual rate (in percents) at which the secured amount will diminish with time, in the worst possible case. This MUST be a number between -100 and 0.^{30 31}

deadline : date-time The prepared transfer can be committed successfully only before this moment. If the client tries to commit the prepared transfer after this moment, the commit MUST NOT be successful.

min_interest_rate : float The value of the **min_interest_rate** field in the corresponding PrepareTransfer message.

ts : date-time The moment at which this message was sent (the message's timestamp).

If a prepared transfer has not been finalized (committed or dismissed) for a long while (1 week for example), the server MUST send another PreparedTransfer message, identical to the previous one (except for the **ts** field), to remind that a transfer has been prepared and is waiting for a resolution. This guarantees that

³⁰Note that when the interest rate on a given account is negative, the secured (locked) amount will be gradually consumed by the accumulated interest. Therefore, at the moment of the prepared transfer's commit, it could happen that the committed amount exceeds the remaining amount by a considerable margin. In such cases, the commit should be unsuccessful. Also, note that when a PrepareTransfer request is being processed by the server, it can not be predicted what amount will be available on the sender's account at the time of the transfer's commit. For this reason, when a PreparedTransfer message is sent, the server should set the value of the **demurrage_rate** field correctly, so as to inform the client (the coordinator) about *the worst possible case*.

Here is an example how this may work, from the viewpoint of a coordinator who is trying to commit a conditional transfer: The coordinator sends a PrepareTransfer message for the conditional transfer, which he knows, because of the still unrealized condition, will take up to 1 month to get finalized. Then, a PreparedTransfer message for this transfer is received, with a **locked_amount** of 1000, and a **demurrage_rate** of -79.5 percent. The coordinator figures out that if he keeps this prepared transfer around, and does not finalize it, for each passed month, up to 2% of the locked amount will be eaten up (0.98 to the power of 12 equals 0.795). Therefore, the coordinator can calculate that in order to be certain that, after one month, he will be able to commit this prepared transfer successfully, the committed amount should not exceed 980. (That is: The value of the **committed_amount** field in the FinalizeTransfer message that the coordinator sends to commit the transfer, should not exceed 980.)

³¹The value of the **demurrage_rate** field in PreparedTransfer messages SHOULD be equal to the most negative interest rate that is theoretically possible to occur on any of the accounts with the given debtor, between the transfer's preparation and the transfer's commit. Note that the current interest rate on the sender's account is not that important, because it can change significantly between the transfer's preparation and the transfer's commit.

prepared transfers will not remain in the server's database forever, even in the case of a lost message, or a complete database loss on the client's side.

FinalizedTransfer

Emitted when a transfer has been finalized (committed or dismissed).

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with **debtor_id** identifies the sender's account.

transfer_id : int64 The opaque ID generated for the prepared transfer. This ID, along with **debtor_id** and **creditor_id**, uniquely identifies the finalized prepared transfer.

coordinator_type : string Indicates the subsystem which requested the transfer. MUST be between 1 and 30 symbols, ASCII only.

coordinator_id : int64 Along with **coordinator_type**, identifies the client that requested the transfer (the *coordinator*).

coordinator_request_id : int64 Along with **coordinator_type** and **coordinator_id**, uniquely identifies the finalized prepared transfer from the coordinator's point of view, so that the coordinator can pair this message with the issued request to finalize the prepared transfer.

committed_amount : int64 The transferred (committed) amount. This MUST always be a non-negative number. A 0 means either that the prepared transfer was dismissed, or that it was committed, but the commit was unsuccessful for some reason.

status_code : string The finalization status. MUST be between 0 and 30 symbols, ASCII only. If the prepared transfer was committed, but the commit was unsuccessful for some reason, this value MUST be different from "OK", and SHOULD hint at the reason for the failure.³² ³³ In all other cases, this value MUST be "OK".

total_locked_amount : int64 When the transfer has been rejected due to insufficient available amount, this field SHOULD contain the total sum

³²The mandatory status codes which MUST be used are:

- "SENDER_IS_UNREACHABLE" signifies that the sender's account does not exist, or can not make outgoing transfers.
- "RECIPIENT_IS_UNREACHABLE" signifies that the recipient's account does not exist, or does not accept incoming transfers.
- "TERMINATED" or anything that starts with "TERMINATED", signifies that the transfer has been terminated due to expired deadline or unapproved interest rate change.
- "TRANSFER_NOTE_IS_TOO_LONG" signifies that the transfer has been rejected because the transfer note's byte-length is too big.
- "INSUFFICIENT_AVAILABLE_AMOUNT" signifies that the transfer has been rejected due to insufficient amount available on the account.

³³When the value of the **status_code** field is different from "OK", the **committed_amount** MUST be zero.

secured (locked) for prepared transfers on the account, *after* this transfer has been finalized. This MUST be a non-negative number.

prepared_at : date-time The moment at which the transfer was prepared.

ts : date-time The moment at which this message was sent (the message's timestamp). This MUST be the moment at which the transfer was committed.

AccountUpdate

Emitted if there has been a meaningful change in the state of an account³⁴, or to remind that an account still exists.

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with **debtor_id**, identifies the account.

creation_date : date The date on which the account was created. Until the account is removed from the server's database, its **creation_date** MUST NOT be changed.³⁵

last_change_ts : date-time The moment at which the latest meaningful change in the state of the account has happened. For a given account, later AccountUpdate messages MUST have later or equal **last_change_tss**, compared to earlier messages.

last_change_seqnum : int32 The sequential number of the latest meaningful change. For a given account, later changes MUST have bigger sequential numbers, compared to earlier changes. Note that when the maximum **int32** value is reached, the next value MUST be -2147483648 (signed 32-bit integer wrapping).^{36 37}

³⁴Every change in the value of one of the fields included in AccountUpdate messages (except for **ts** and **ttl** fields) should be considered meaningful, and therefore an AccountUpdate message MUST *eventually* be emitted to inform about the change. There is no requirement, though, AccountUpdate messages to be emitted instantly, following each individual change. For example, if a series of transactions are committed on an account in a short period of time, the server SHOULD emit only one AccountUpdate message, announcing only the final state of the account. As a rough guideline, on average, AccountUpdate messages for one account should not be sent more often than once per hour.

³⁵Note that an account can be removed from the server's database, and then a new account with the same **debtor_id** and **creditor_id** can be created. In those cases care MUST be taken, so that the newly created account always has a later **creation_date**, compared to the preceding account. The most straightforward way to achieve this is not to remove accounts on the same day on which they have been created.

³⁶**creation_date**, **last_change_ts**, and **last_change_seqnum** can be used to reliably determine the correct order in a sequence of AccountUpdate messages, even if the changes occurred in a very short period of time. When considering two changes, **creation_date** fields MUST be compared first, if they are equal **last_change_ts** fields MUST be compared, and if they are equal, **last_change_seqnum** fields MUST be compared as well.

³⁷Note that when comparing "seqnum" fields, server implementations MUST correctly deal with the possible 32-bit integer wrapping. For example, to decide whether **seqnum2** is later than **seqnum1**, the following expression may be used: $0 < (\text{seqnum2} - \text{seqnum1}) \% 0x100000000 <$

principal : int64 The amount that the debtor owes to the creditor, without the interest. This can be a negative number.

interest : float The amount of interest accumulated on the account up to the **last_change_ts** moment, which is not added to the **principal** yet. Once in a while, the accumulated interest MUST be zeroed out and added to the principal (an interest payment). Note that the accumulated interest can be a negative number, but MUST be *finite*.³⁸

interest_rate : float The annual rate (in percents) at which interest accumulates on the account. This can be a negative number, but MUST NOT be smaller than -100, and MUST be *finite*.

last_interest_rate_change_ts : date-time The moment at which the latest change in the account's interest rate happened. For a given account, later AccountUpdate messages MUST have later or equal **last_interest_rate_change_tss**, compared to earlier messages. The minimum time interval between two changes in the account's interest rate MUST be big enough so as to provide a reasonable guarantee that, even in case of a temporary network disconnect, at least 24 hours have passed since the AccountUpdate message sent for the previous interest rate change has been processed by all clients. If there have not been any changes in the interest rate yet, the value MUST be "1970-01-01T00:00:00+00:00".

last_config_ts : date-time MUST contain the value of the **ts** field in the latest applied ConfigureAccount message. If there have not been any applied ConfigureAccount messages yet, the value MUST be "1970-01-01T00:00:00+00:00".

last_config_seqnum : int32 MUST contain the value of the **seqnum** field in the latest applied ConfigureAccount message. If there have not been any applied ConfigureAccount messages yet, the value MUST be 0.³⁹

negligible_amount : float The value of the **negligible_amount** field in the latest applied ConfigureAccount message. If there have not been any applied ConfigureAccount messages yet, the value MUST represent the current configuration settings. This MUST always be a *finite* non-negative number.

config_flags : int32 The value of the **config_flags** field in the latest applied ConfigureAccount message. If there have not been any applied ConfigureAccount messages yet, the value MUST represent the current configuration settings.

0x80000000. Timestamps must also be compared with care, because precision might have been lost when they were saved to the database.

³⁸The accumulated interest MUST be available for transfers. That is: the owner of the account has to be able to "wire" the accumulated interest to another account. Accordingly, accumulated negative interest MUST be subtracted from the account's available amount.

³⁹Note that clients can use **last_config_ts** and **last_config_seqnum** to determine whether a sent ConfigureAccount message has been applied successfully.

config_data : string The value of the **config_data** field in the latest applied ConfigureAccount message. If there have not been any applied ConfigureAccount messages yet, the value MUST represent the current configuration settings.⁴⁰

account_id : string A string which (along with **debtor_id**) publicly and globally identifies the account.⁴¹ An empty string indicates that the account does not have an identity yet.⁴² Once the account have got an identity, the identity SHOULD NOT be changed until the account is removed from the server's database.

debtor_info_iri : string A link (Internationalized Resource Identifier) for obtaining information about the account's debtor. This provides a reliable way for creditors to get up-to-date information about the debtor. Note that changing the IRI will likely cause the clients to make requests to the new IRI, so as to obtain updated information about the debtor. The link MUST have at most 200 Unicode characters. If no link is available, the value SHOULD be an empty string.

debtor_info_content_type : string The content type of the document that the **debtor_info_iri** link refers to. It MUST have at most 100 symbols, ASCII only. If no link is available, or the content type of the document is unknown, the value SHOULD be an empty string.

debtor_info_sha256 : bytes The SHA-256 cryptographic hash of the content of the document that the **debtor_info_iri** link refers to. MUST contain exactly 0, or exactly 32 bytes. If no link is available, or the SHA-256 cryptographic hash of the document is unknown, the value SHOULD contain 0 bytes.

last_transfer_number : int64 MUST contain the value of the **transfer_number** field in the latest emitted AccountTransfer message for the account. If since the creation of the account there have not been any emitted AccountTransfer messages, the value MUST be 0.

last_transfer_committed_at : date-time MUST contain the value of the **committed_at** field in the latest emitted AccountTransfer message for the account. If since the creation of the account there have not been any emitted AccountTransfer messages, the value MUST be "1970-01-01T00:00:00+00:00".

demurrage_rate : float The demurrage rate (in percents) for new prepared transfers. That is: the value of the **demurrage_rate** field in new Prepared-

⁴⁰The UTF-8 encoding of the **config_data** string MUST NOT be longer than 2000 bytes.

⁴¹The account identifier MUST have at most 100 symbols, ASCII only. Different server implementations may use different formats for this identifier. Note that **creditor_id** is an ID which is known only to the client that created the account. The account identifier (along with **debtor_id**), on the other hand, MUST provide enough information to publicly and globally identify the account (an IBAN for example).

⁴²When the account does not have an identity, it can not accept incoming transfers.

Transfer messages. This MUST be a number between -100 and 0, which SHOULD be the same for all accounts with the given debtor.⁴³

commit_period : int32 The maximal allowed period (in seconds) during which new prepared transfers can be committed successfully. That is: unless the client explicitly requested the deadline for the transfer to be shorter than normal, the value of the **deadline** field in new PreparedTransfer messages will be calculated by adding **commit_period** seconds to the **prepared_at** timestamp. The value of this field MUST be a non-negative number, SHOULD be the same for all accounts with the given debtor, and SHOULD be at least 86400 (24 hours).

transfer_note_max_bytes: int32 The maximal number of bytes that the **transfer_note** field in FinalizeTransfer messages is allowed to contain when UTF-8 encoded. This MUST be a non-negative number which does not exceed the general limit imposed by this protocol.⁴⁴ When changed, it SHOULD NOT be decreased.

ts : date-time The moment at which this message was sent (the message's timestamp).

ttl : int32 The time-to-live (in seconds) for this message. The message SHOULD be ignored if more than **ttl** seconds have elapsed since the message was emitted (**ts**). This MUST be a non-negative number.

If for a given account, no AccountUpdate messages have been sent for a long while (1 week for example), the server MUST send a new AccountUpdate message identical to the previous one (except for the **ts** field), to remind that the account still exist. This guarantees that accounts will not remain in the server's database forever, even in the case of a lost message, or a complete database loss on the client's side. Also, this serves the purpose of a "heartbeat", allowing clients to detect "dead" account records in their databases.

AccountPurge

Emitted some time after an account has been removed from the server's database.⁴⁵

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with **debtor_id**, identifies the removed account.

⁴³The value of the **demurrage_rate** field in PreparedTransfer messages SHOULD be equal to the most negative interest rate that is theoretically possible to occur on any of the accounts with the given debtor, between the transfer's preparation and the transfer's commit. Note that the current interest rate on the sender's account is not that important, because it can change significantly between the transfer's preparation and the transfer's commit.

⁴⁴The UTF-8 encoding of the **transfer_note** string MUST NOT be longer than 500 bytes.

⁴⁵The AccountPurge message delay MUST be long enough to ensure that after clients have received the AccountPurge message, if they continue to receive old, wandering AccountUpdate messages for the purged account, those messages will be ignored (due to expired **ttl**).

creation_date : date The date on which the removed account was created.

ts : date-time The moment at which this message was sent (the message's timestamp).

The purpose of AccountPurge messages is to inform clients that they can safely remove a given account from their databases.

AccountTransfer

Emitted when a non-negligible committed transfer has affected a creditor's account.⁴⁶ ⁴⁷

debtor_id : int64 The ID of the debtor.

creditor_id : int64 Along with **debtor_id**, identifies the affected account.

creation_date : date The date on which the affected account was created.

transfer_number : int64 Along with **debtor_id**, **creditor_id**, and **creation_date**, uniquely identifies the non-negligible committed transfer. This **MUST** be a positive number. During the lifetime of a given account, later committed transfers **MUST** have bigger **transfer_numbers**, compared to earlier transfers.⁴⁸

coordinator_type : string Indicates the subsystem which requested the transfer. **MUST** be between 1 and 30 symbols, ASCII only.

⁴⁶A *negligible transfer* is an incoming transfer for which the transferred amount does not exceed the **negligible_amount** configured for the recipient's account (that is: $0 < \text{acquired_amount} \leq \text{negligible_amount}$).

⁴⁷To issue new tokens into existence, the server **SHOULD** use a special account called "*the debtor's account*" (or "the root account"). The debtor's account is special in the following ways:

- The **creditor_id** for the debtor's account is 0.
- The balance on the debtor's account is allowed to go negative, as long as it does not exceed the configured **negligible_amount** for the account (with a negative sign). This gives debtors agents the option to reliably restrict the total amount that a debtor is allowed to issue.
- Interest is not accumulated on the debtor's account.
- All interest payments to/from creditor's accounts, come from/to the debtor's account.
- AccountTransfer messages are not sent for transfers from/to the debtor's account. This eliminates a potentially huge amount of network traffic towards the debtor's account, especially for interest payments.
- Each debtor can use its debtor's account **config_data** text field, to configure various important parameters of the currency (like the interest rate). The format for the **config_data** text field will be specified in separate document(s).
- The debtor's account should always be able to receive incoming transfers, even if it does not exist yet, or is "scheduled for deletion". Transferring money to the debtor's account is equivalent to "destroying" the money.

⁴⁸Note that when an account has been removed from the database, and then recreated again, the generation of transfer numbers **MAY** start from 1 again.

sender : string A string which (along with **debtor_id**) identifies the sender's account.⁴⁹ An empty string signifies that the sender is unknown.

recipient : string A string which (along with **debtor_id**) identifies the recipient's account.⁵⁰ An empty string signifies that the recipient is unknown.

acquired_amount : int64 The increase in the affected account's principal (caused by the transfer). This MUST NOT be zero. If it is a positive number (an addition to the principal), the affected account would be the recipient. If it is a negative number (a subtraction from the principal), the affected account would be the sender.

transfer_note : string If the transfer has been committed by a `FinalizeTransfer` message, this field MUST contain the value of the **transfer_note** field from the message that committed the transfer. Otherwise, it SHOULD contain information pertaining to the reason for the transfer.⁵¹

transfer_note_format : string If the transfer has been committed by a `FinalizeTransfer` message, this field MUST contain the value of the **transfer_note_format** field from the message that committed the transfer. Otherwise, it MUST contain the format used for the **transfer_note** string.⁵²

committed_at : date-time The moment at which the transfer was committed.

principal : int64 The amount that the debtor owes to the owner of the affected account, without the interest, after the transfer has been committed. This can be a negative number.

ts : date-time The moment at which this message was sent (the message's timestamp).

previous_transfer_number : int64 MUST contain the **transfer_number** of the previous `AccountTransfer` message that affected the same account. If since the creation of the account, there have not been any other committed transfers that affected it, the value MUST be 0.

⁴⁹The account identifier MUST have at most 100 symbols, ASCII only. Different server implementations may use different formats for this identifier. Note that **creditor_id** is an ID which is known only to the client that created the account. The account identifier (along with **debtor_id**), on the other hand, MUST provide enough information to publicly and globally identify the account (an IBAN for example).

⁵⁰The account identifier MUST have at most 100 symbols, ASCII only. Different server implementations may use different formats for this identifier. Note that **creditor_id** is an ID which is known only to the client that created the account. The account identifier (along with **debtor_id**), on the other hand, MUST provide enough information to publicly and globally identify the account (an IBAN for example).

⁵¹The UTF-8 encoding of the **transfer_note** string MUST NOT be longer than 500 bytes.

⁵²The value of the **transfer_note_format** field MUST match the regular expression `^[0-9A-Za-z.-]{0,8}$`.

Every committed transfer affects two accounts: the sender's, and the recipient's. Therefore, two separate AccountTransfer messages would be emitted for each committed non-negligible transfer.

Requirements for Client Implementations

RT record

Before sending a PrepareTransfer message, client implementations **MUST** create a *running transfer record* (RT record) in the client's database, to track the progress of the requested transfer. The primary key for running transfer records is the (coordinator_type, coordinator_id, coordinator_request_id) tuple. As a minimum, RT records **MUST** also be able to store the values of debtor_id, creditor_id, and transfer_id fields. RT records **MUST** have 3 possible statuses:

initiated Indicates that a PrepareTransfer request has been sent, and no response has been received yet. RT records with this status **MAY** be deleted whenever considered appropriate. Newly created records **MUST** receive this status.

prepared Indicates that a PrepareTransfer request has been sent, and a PreparedTransfer response has been received. RT records with this status **MUST NOT** be deleted. Instead, they need to be settled first (committed or dismissed), by sending a FinalizeTransfer message.⁵³

settled Indicates that a PrepareTransfer request has been sent, a PreparedTransfer response has been received, and a FinalizeTransfer message has been sent to dismiss or commit the transfer. RT records for *dismissed transfers* **MAY** be deleted whenever considered appropriate. RT records for *committed transfers*, however, **SHOULD NOT** be deleted right away. Instead, they **SHOULD** stay in the database until a FinalizedTransfer message is received for them, or a very long time has passed.^{54 55 56}

⁵³If a "prepared" RT record is lost due to a database crash, after some time (possibly a long time) a PreparedTransfer message will be received again for the transfer, and the transfer will be dismissed by the client. This must not be allowed to happen regularly, because it would cause the server to keep the prepared transfer locks for much longer than necessary.

⁵⁴The retention of committed RT records is necessary to prevent problems caused by message re-delivery. Consider the following scenario: a transfer has been prepared and committed (settled), but the PreparedTransfer message is re-delivered a second time. Had the RT record been deleted right away, the already committed transfer would be dismissed the second time, and the fate of the transfer would be decided by the race between the two different finalizing messages. In most cases, this would be a serious problem.

⁵⁵That is: if the corresponding FinalizedTransfer message has not been received for a very long time (1 year for example), the RT record for the committed transfer **MAY** be deleted, nevertheless.

⁵⁶Note that FinalizedTransfer messages are emitted for dismissed transfers as well. Therefore, the most straightforward policy is to delete RT records for both committed and dismissed transfers the same way.

Received RejectedTransfer message

When client implementations process a RejectedTransfer message, they should first try to find a matching RT record in the client's database.⁵⁷ If a matching record exists, and its status is "initiated", the transfer can be reported as unsuccessful, and the RT record MAY be deleted; otherwise the message SHOULD be ignored.

Received PreparedTransfer message

When client implementations process a PreparedTransfer message, they MUST first try to find a matching RT record in the client's database.⁵⁸ If a matching record does not exist, the newly prepared transfer MUST be immediately dismissed⁵⁹; otherwise, the way to proceed depends on the status of the RT record:

initiated The values of `debtor_id`, `creditor_id`, and `transfer_id` fields in the received PreparedTransfer message MUST be stored in the RT record, and the status of the record MUST be set to "prepared".

prepared The values of `debtor_id`, `creditor_id`, and `transfer_id` fields in the received PreparedTransfer message MUST be compared to the values stored in the RT record. If they are the same, no action SHOULD be taken; if they differ, the newly prepared transfer MUST be immediately dismissed.⁶⁰

settled The values of `debtor_id`, `creditor_id`, and `transfer_id` fields in the received PreparedTransfer message MUST be compared to the values stored in the RT record. If they are the same, the same FinalizeTransfer message (except for the `ts` field), which was sent to finalize the transfer, MUST be sent again; if they differ, the newly prepared transfer MUST be immediately dismissed.⁶¹

Important note: Eventually a FinalizeTransfer message MUST be sent for each "prepared" RT record, and the record's status set to "settled". Often this can be done immediately. In this case, when the PreparedTransfer message is

⁵⁷The matching RT record MUST have the same `coordinator_type`, `coordinator_id`, and `coordinator_request_id` values as the received PreparedTransfer message. Additionally, the values of other fields in the received message MAY be verified as well, so as to ensure that the server behaves as expected.

⁵⁸The matching RT record MUST have the same `coordinator_type`, `coordinator_id`, and `coordinator_request_id` values as the received PreparedTransfer message. Additionally, the values of other fields in the received message MAY be verified as well, so as to ensure that the server behaves as expected.

⁵⁹A prepared transfer is dismissed by sending a FinalizeTransfer message, with zero `committed_amount`.

⁶⁰A prepared transfer is dismissed by sending a FinalizeTransfer message, with zero `committed_amount`.

⁶¹A prepared transfer is dismissed by sending a FinalizeTransfer message, with zero `committed_amount`.

received, the matching RT record will change its status from "initiated", directly to "settled".

Received FinalizedTransfer message

When client implementations process a FinalizedTransfer message, they should first try to find a matching RT record in the client's database.⁶² If a matching record exists, its status is "settled", and the values of `debtor_id`, `creditor_id`, and `transfer_id` fields in the received message are the same as the values stored in the RT record, then the outcome of the finalized transfer can be reported, and the RT record MAY be deleted; otherwise the message SHOULD be ignored.

AD record

Client implementations *that manage creditor accounts*, MUST maintain *account data records* (AD records) in their databases, to store accounts' current status data. The primary key for account data records is the (`creditor_id`, `debtor_id`, `creation_date`) tuple.⁶³ As a minimum, AD records MUST also be able to store the values of `last_change_ts` and `last_change_seqnum` fields from the latest received AccountUpdate message, plus they SHOULD have a `last_heartbeat_ts` field.⁶⁴

Received AccountUpdate message

When client implementations process an AccountUpdate message, they should first verify message's `ts` and `ttl` fields. If the message has "expired", it SHOULD be ignored. Otherwise, implementations MUST verify whether a corresponding AD record already exists:⁶⁵

1. If a corresponding AD record already exists, the value of its `last_heartbeat_ts` field SHOULD be advanced to the value of the `ts` field in the received message.⁶⁶ Then it MUST be verified whether the same or a later AccountUpdate message has been received already.⁶⁷

⁶²The matching RT record MUST have the same `coordinator_type`, `coordinator_id`, and `coordinator_request_id` values as the received PreparedTransfer message. Additionally, the values of other fields in the received message MAY be verified as well, so as to ensure that the server behaves as expected.

⁶³Another, probably more practical alternative, is the primary key for AD records to be the (`creditor_id`, `debtor_id`) tuple. In this case, later `creation_dates` should simply override earlier `creation_dates`.

⁶⁴The AD record's `last_heartbeat_ts` field stores the timestamp of the latest received account heartbeat.

⁶⁵The corresponding AD record would have the same values, as in the received message, for the fields included in the record's primary key.

⁶⁶That is: the value of the `last_heartbeat_ts` field SHOULD be changed only if the value of the `ts` field in the received AccountUpdate message represents a later timestamp. Also, care SHOULD be taken to ensure that the new value of `last_heartbeat_ts` is not far in the future, which can happen if the server is not behaving correctly.

⁶⁷`creation_date`, `last_change_ts`, and `last_change_seqnum` can be used to reliably determine the correct order in a sequence of AccountUpdate messages, even if the changes occurred

⁶⁸ If the received message turns out to be an old one, further actions MUST NOT be taken; otherwise, the corresponding AD record MUST be updated with the data contained in the received message.

2. If a corresponding AD record does not exist, one of the following two actions MUST be taken: either a new AD record is created, or a ConfigureAccount message is sent to schedule the account for deletion.⁶⁹

If for a given account, AccountUpdate messages have not been received for a very long time (1 year for example), the account's AD record MAY be removed from the client's database.⁷⁰

Received AccountPurge message

When client implementations process an AccountPurge message, they should first verify whether an AD record exists, which has the same values for `creditor_id`, `debtor_id`, and `creation_date` as the received message. If such AD record exists, it SHOULD be removed from the client's database; otherwise, the message SHOULD be ignored.

AL record

Client implementations MAY maintain *account ledger records* (AL records) in their databases, to store accounts' transfer history data. The main function of AL records is to reconstruct the original order in which the processed AccountTransfer messages were sent.⁷¹ The primary key for account ledger records is the (`creditor_id`, `debtor_id`, `creation_date`) tuple. As a minimum, AL records must also be able to store a set of processed AccountTransfer messages, plus a `last_transfer_number` field, which contains the transfer number of the latest

in a very short period of time. When considering two changes, `creation_date` fields MUST be compared first, if they are equal `last_change_ts` fields MUST be compared, and if they are equal, `last_change_seqnum` fields MUST be compared as well.

⁶⁸Note that when comparing "seqnum" fields, server implementations MUST correctly deal with the possible 32-bit integer wrapping. For example, to decide whether `seqnum2` is later than `seqnum1`, the following expression may be used: `0 < (seqnum2 - seqnum1) % 0x100000000 < 0x80000000`. Timestamps must also be compared with care, because precision might have been lost when they were saved to the database.

⁶⁹In this case, the `negligible_amount` field MUST be a set to some huge number, so as to ensure that the account will be successfully deleted by the server.

⁷⁰The AD record's `last_heartbeat_ts` field stores the timestamp of the latest received account heartbeat.

⁷¹Note that AccountTransfer messages can be processed out-of-order. For example, it is possible *transfer #3* to be processed right after *transfer #1*, and only then *transfer #2* to be received. In this case, *transfer #3* should not be added to the account's ledger before *transfer #2* has been processed as well. Thus, in this example, the value of `last_transfer_number` will be updated from 1 to 3, but only after *transfer #2* has been processed successfully.

An important case which client implementations should be able to deal with is when, in the previous example, *transfer #2* is never received (or at least not received for a quite long time). In this case, the AL record should to be "patched" with a made-up transfer, so that the record remains consistent, and can continue to receive transfers.

transfer that has been added to the given account's ledger.⁷²

Received AccountTransfer message

When client implementations process an AccountTransfer message, they must first verify whether a corresponding AL record already exists.⁷³ If it does not exist, a new AL record may be created.⁷⁴ Then, if there is a corresponding AL record (an already existing one, or the one that have been just created), the following steps must be performed:

1. The received message must be added to the set of processed AccountTransfer messages, stored in the corresponding AL record.
2. If the value of the `previous_transfer_number` field in the received message is the same as the value of the `last_transfer_number` field in the corresponding AL record, the `last_transfer_number`'s value must be updated to contain the transfer number of the *latest sequential transfer* in the set of processed AccountTransfer messages.⁷⁵ ⁷⁶ Note that when between two AccountTransfer messages that are being added to the ledger, there were one or more negligible transfers, a dummy in-between ledger entry must be added as well, so as to compensate for the negligible transfers (for which AccountTransfer messages have not been sent).

Note: Client implementations should have some way to remove created AL records that are not needed anymore.

⁷²Note that AccountTransfer messages form a singly linked list. That is: the `previous_transfer_number` field in each message refers to the value of the `transfer_number` field in the previous message.

⁷³The corresponding AL record would have the same values for `creditor_id`, `debtor_id`, and `creation_date` as the received AccountTransfer message.

⁷⁴The newly created AL record must have the same values for `creditor_id`, `debtor_id`, and `creation_date` as the received AccountTransfer message, an empty set of stored AccountTransfer messages, and a `last_transfer_number` field with the value of 0.

⁷⁵Note that AccountTransfer messages can be processed out-of-order. For example, it is possible *transfer #3* to be processed right after *transfer #1*, and only then *transfer #2* to be received. In this case, *transfer #3* should not be added to the account's ledger before *transfer #2* has been processed as well. Thus, in this example, the value of `last_transfer_number` will be updated from 1 to 3, but only after *transfer #2* has been processed successfully.

An important case which client implementations should be able to deal with is when, in the previous example, *transfer #2* is never received (or at least not received for a quite long time). In this case, the AL record should be "patched" with a made-up transfer, so that the record remains consistent, and can continue to receive transfers.

⁷⁶Note that AccountTransfer messages form a singly linked list. That is: the `previous_transfer_number` field in each message refers to the value of the `transfer_number` field in the previous message.