3.2.1. Architecture of the synchronization module

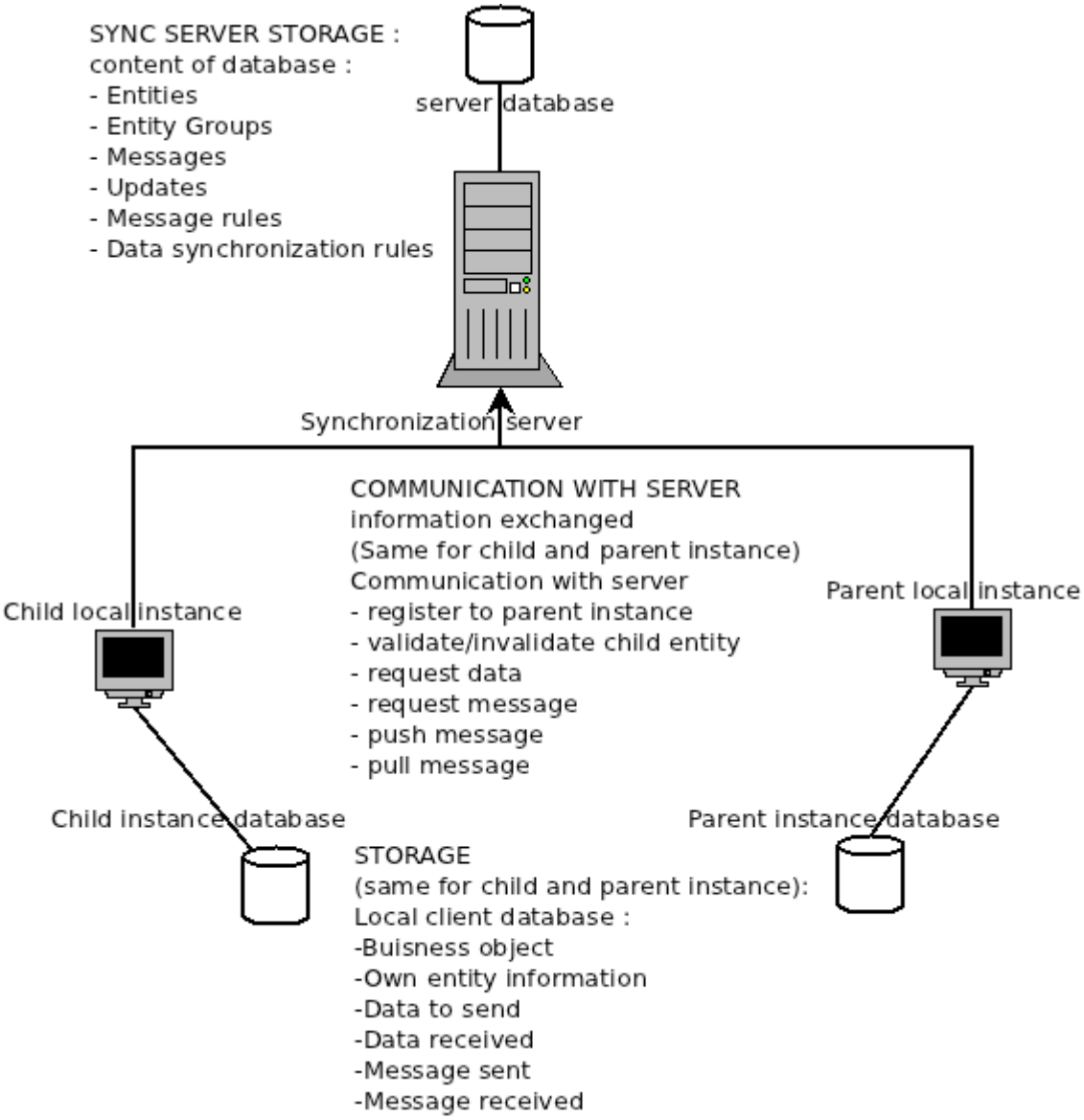
3.2.1.1 Definitions

- **set :** unordered list of elements without repeated elements
- **model :** the abstract description of a type of record, eg : Sales order, Purchase order, Partner. A type of record.
- **record :** instance of a model (eg : The purchase order 238). From a database point of view, it corresponds to a line in the table.
- **Openerp instance :** An OpenERP instance is an OpenERP server running with its own database, it can run offline and independently of any other server.
- **Instance:** MSF logical organisation part : Cell, Coordination, project... An instance can have children and one parent. Each Instance runs its own OpenERP application server.
- **Group:** A set of instance, represents a mission, a section... There is a type associated with each group : Section, Coordination, International
- **Data transfer and message transfer :**
    - Data transfer copies only information existing in an instance to another instance.
      Use data copy to find the same record on different instances,
      e.g. copy information about a supplier
    - Message transfer is used to communicate between two instances when one action in one instance has a consequence on the other. Message are used to interact in more complex way with other instances, e.g. purchase order for an internal supplier (another mission) will create a sale order in the instance of this supplier. One instance has the purchase order, the other instance has the sale order.
    - Data transfer allows synchronizing data between many instances, message transfer just involve two instances: the sender and the receiver.
- **Update:** An update is a record that represents a modification on a given record, synchronization server store all the modification to synchronize using that kind of record.
- **Xml_id**: This is the name given to the id of data that comes from another data source. In this project this name is the unique id of a record among all the OpenERP instance.

3.2.1.2 General description

The POC implements entity management, data synchronization with session management and asynchronous messages passing. This document gives solution for error monitoring and error recovery, for conflict detection and conflict management and finally gives some idea for the implementation of a physical synchronization and crash recovery.

This schema shows a high level representation of the synchronization system. It shows the unique server with its database and two local instances (client) with their own database. It shows which type of record are stored client side and server side and also what kind of request client sends to the server.



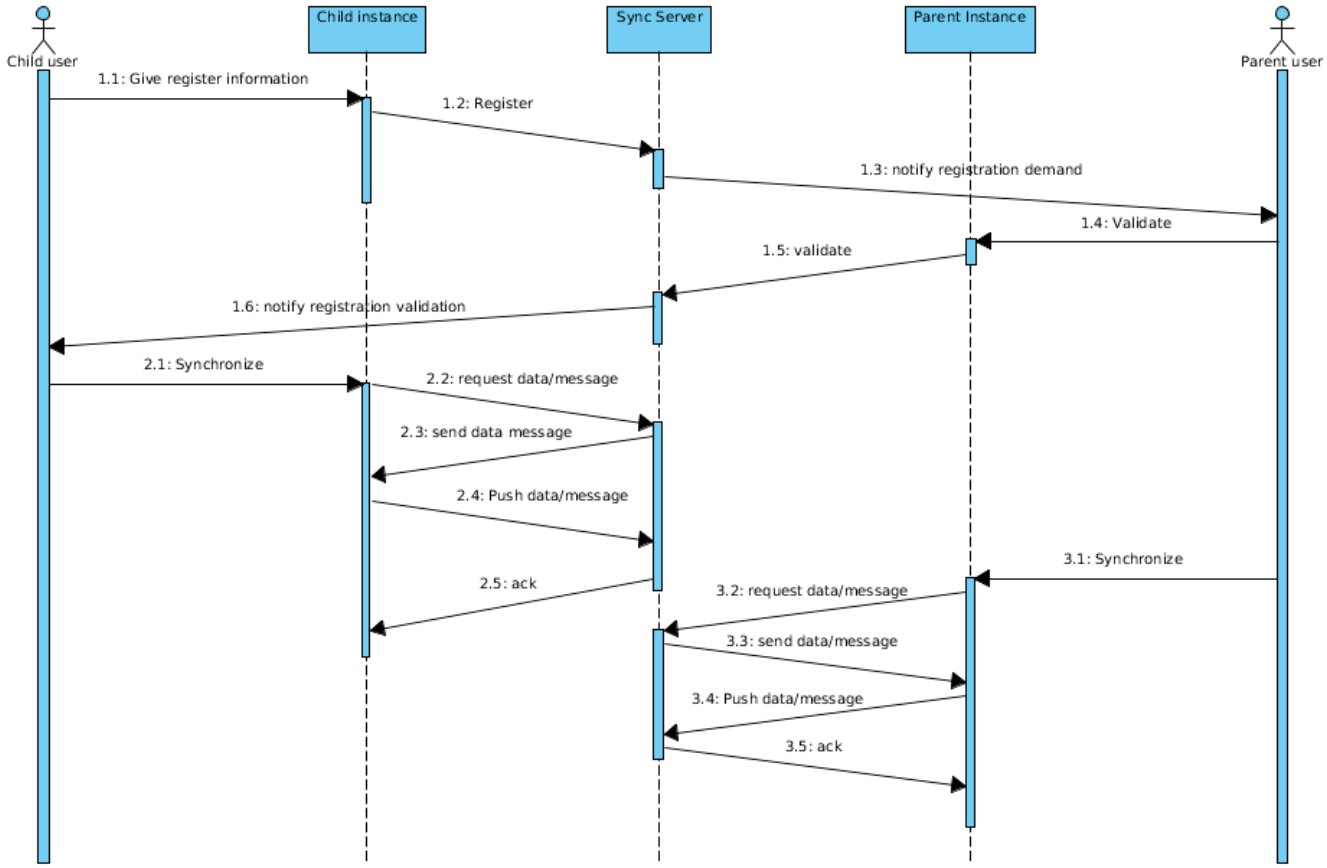
The next schema shows how client and server communicate. The child has to register itself on the server specifying who its parent is, and which groups it is member of. That information is important to reproduce the organization of MSF and to know which entities will synchronize which data with which other entities. The child client cannot start its synchronization until the declaring parent validated its status on the server.

Child and parent instance have no difference in their storage or their communication with the server. Usually parent instance is also a child of another instance, and child instance can have its own children.

This scenario is a success case where everything goes well. Each process is detailed further in this document. This diagram shows three different operations that may happen in this order:

1. the first operation is the all registration process,
2. the second operation is the synchronization of the new registered instance
3. and the third operation is the synchronization of the parent instance (but could be any instance of the system).

Of course the order of operation 1, 2, 3 can be completely different; the only constraint is that 1 must come before 2.

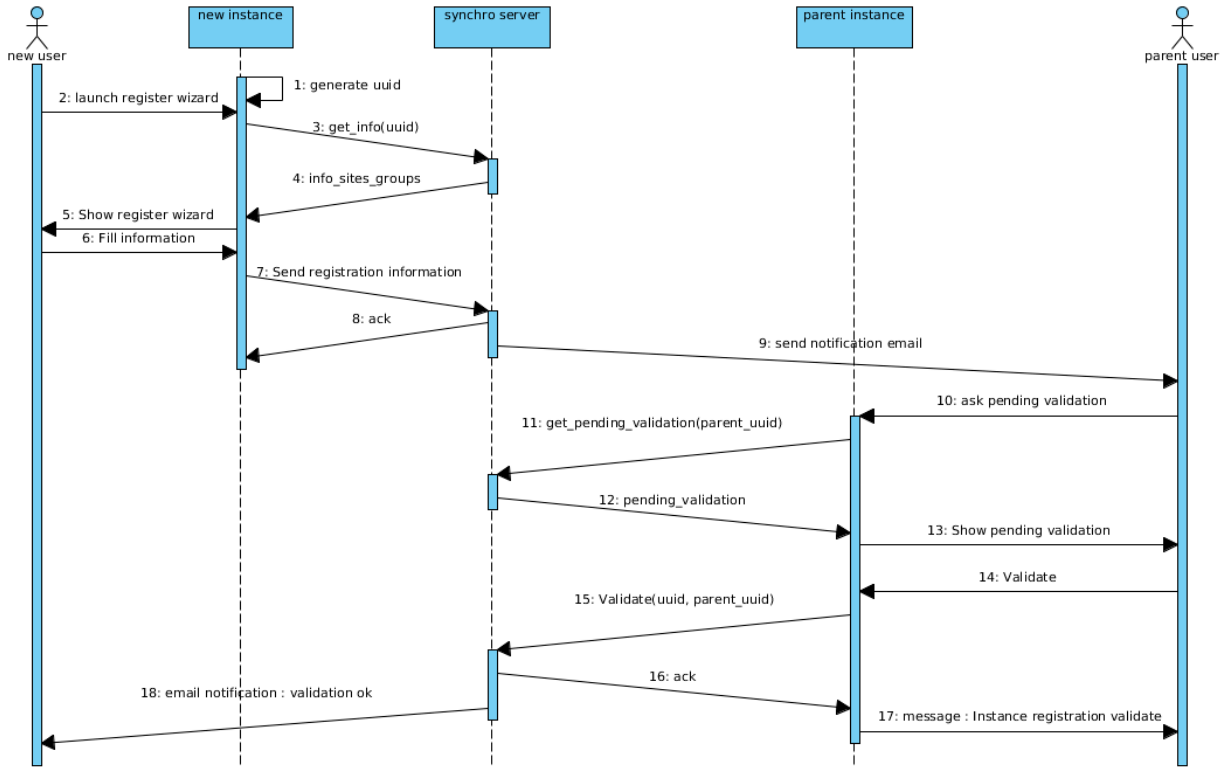


3.2.2. Identification of instances and groups

Each instance has its unique local instance of OpenERP. The instance is identified by the instance of OpenERP, and the instance is identified by a unique id. The unique generate by the standart lib of python uuid1. The unique id is created by the client and communicated to the server during the registration process.

In the scope of this POC, the registration wizard will allows an entity to request another entity to be its parent as well as to be part of groups. This will have to be approved by the selecting parent. The entity cannot synchronize data or messages before this approval.
In the scope of this POC, we allow any ancestor of the instance to validate it. Ancestor is any entity at higher node in the hierarchy tree.

Note. The notion of parent defines the tree structure among the entities. This structure will impact the synchronization with a given entity according to the directionality of the synchronization rules. For example, a synchronization rule applied to a model A, to a group Mission 1 with directionality "Down" will result in the synchronization of all instances of A created in the coordination to all its children projects, but not the other way round.



The local instances only store the uuid, the name and the parent's name but do not store the group they belong to nor the ids of other entities. The server stores the id of all entities and the hierarchy.

Security

To synchronize, an instance need to give is uuid and this uuid has to be validated. But in order to connect to an OpenERP application server you have to give a user id and the password for this user. Those information are store in the connection manager. But the password is not store in the database but in the memory of the server. In this way, the client instance cannot connect unless the admin has set the password after launching the server. If someone steals a backup or the server, he will not be able to synchronize again because he does not have the password.

Invalidation

An instance is able to invalidate any children instance or progeny. Each invalidation will be described by a short note sent by email to all invalidated instances.
Child instances can change their registration information at any time but if they do so, they will be invalidated by the server and their new parent should approve this change anyway.

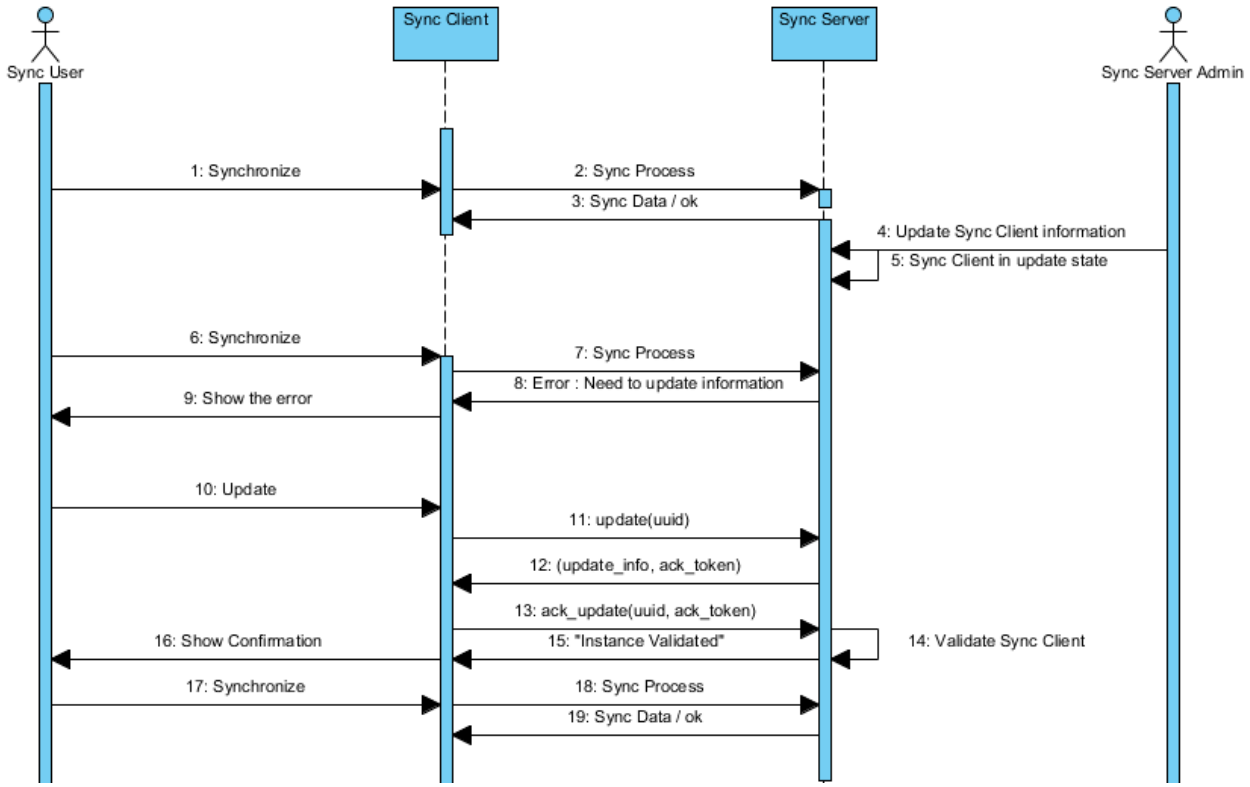
If a parent do not validate a child due to wrong registration information (wrong name, wrong group or wrong parent), the parent should notifiy manually child by email and once a child instance has modified wrong information, the parent instance user is notified and can approve the modification.

3.2.2.1 Structure and entity management server side

The administrator of the synchronization server, can modify any entities information (manage groups ownership, hierarchy), create new entities but unique id cannot be set server side.
But in order to keep data consistency, in case of modification, existing entities should be notified of the change and request information update. Entities with information modified cannot synchronize data anymore until they request update of information and send an acknowledgement (ACK) to the server.

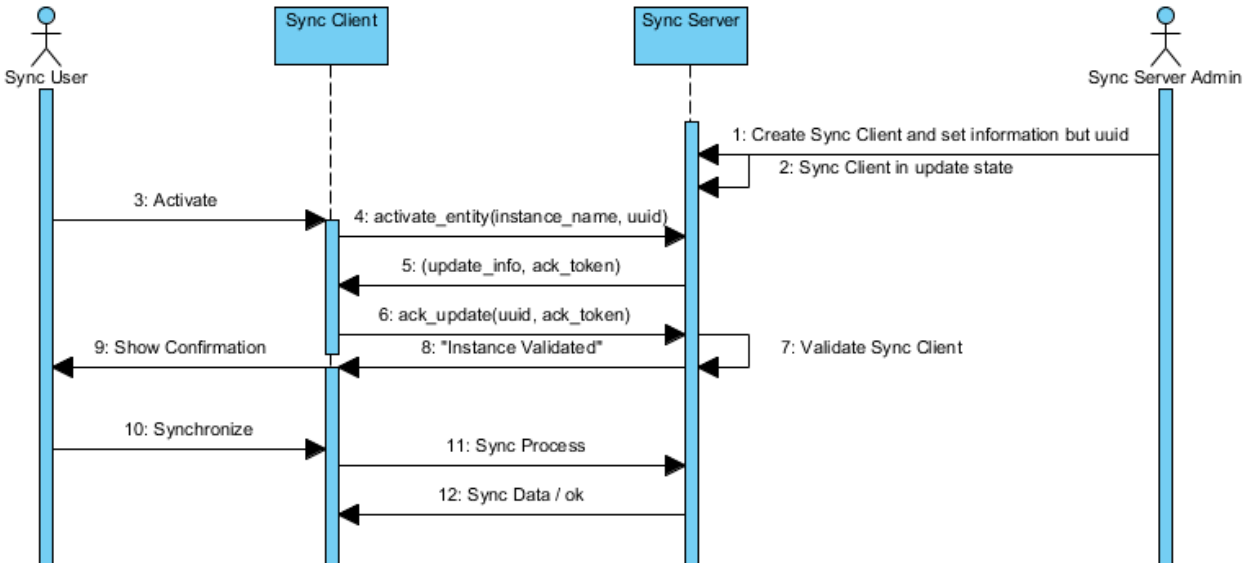
When entities are created server side, the local instance has to run the activation process to set the unique id and in this case the instance doesn’t need to be validated by the parent.

Detail of information update process



This Diagram shows in details what happens when the administrator of the sync server update information about an instance (sync client). Automatically, the server set the instance in update state. It means the instance will get an error (Need to update information) if it try to synchronize and to start the update process. The information update process consist in two request, the first one is the request for the information and the second is an ack to notify the server that the client as receive and saved the new information. In order to avoid instance just sending the ack (and then be validated with outdated data) a token is given by the server in the response of the first request and has to be given in the second request.

Detail of instance activation process



The activation process is quite similar to the update process. The only difference is the first request. In the activate process, the server does not have the uuid of the client, it has only the name, so to be activated the client instance has to give the name (chosen by the sync server administrator) and the uuid generated by the client instance.

3.2.3. Data transfer

The transfer of data is always done through the unique synchronization server. It is bidirectional, asynchronous and performed by two distinct operations:

- the push of data that sends local modifications (updates) to the server for sharing with other entities
- the pull of data that retrieve modifications (updates) done by other instances

Updates are generated according to synchronization rules. A rule specifies a model (to select the working table) and a domain defining which records (which row of the table) of a model have to be synchronized. A rule specifies which fields of the records (which columns of the table) have to be pushed and specifies for the pull operation forced values and/or fallback values for many2one relations in case the local instance does not have the linked record. The rule defines which group of entity or type of entity will push data and in which direction (children, parent) the data has to be transferred.

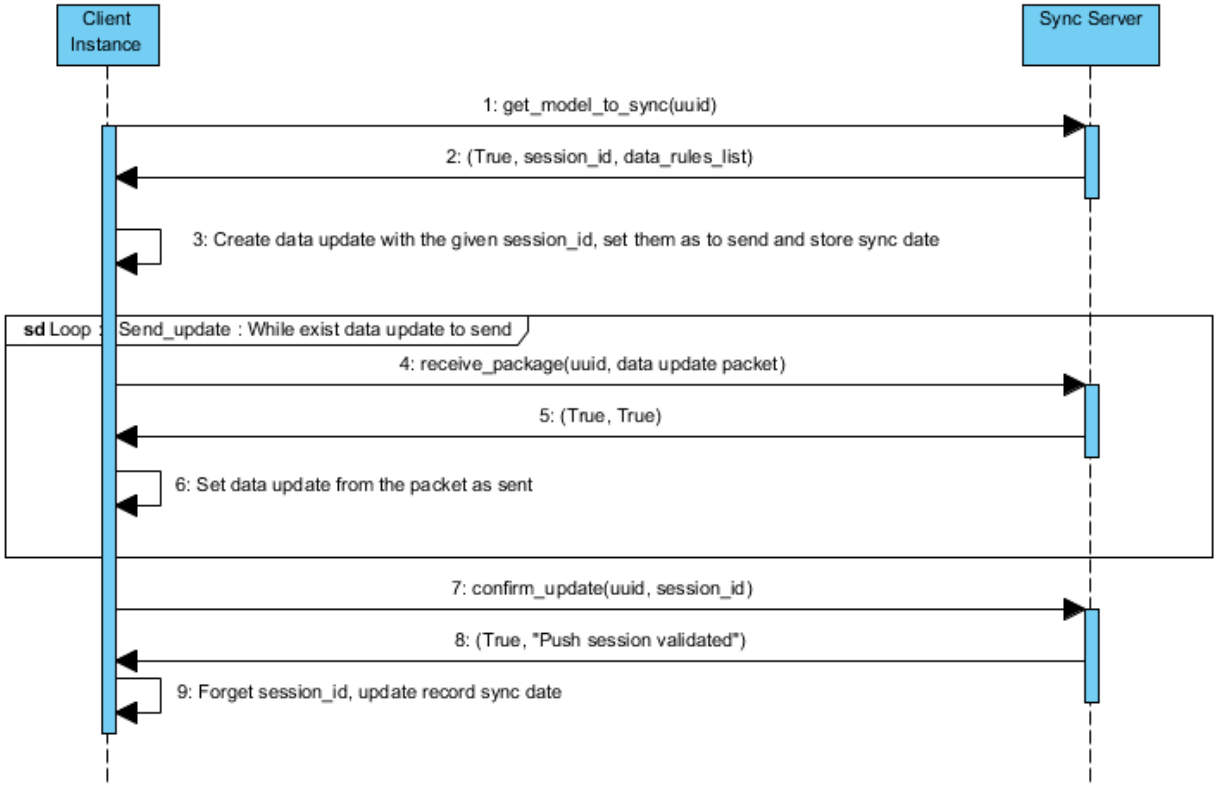
3.2.3.1. Pushing data/updates

Here is the summary of the normal process of pushing data.

1. The client requests a new push session on the server and retrieves back the concerned rules in order to know which data to push and the session ID.
2. It prepares the updates to send and creates packets.
3. It sends the packets one by one to the server with the given session ID.
4. It sends validation for the session, once the ACK of the validation is received, closes the session.

An instance can run only one push session at a time. The session must be ended before starting a new one.

The following diagram describes the normal process:



The push is triggered by the user (or a scheduled task) on the local instance.

1. The local instance requests the server to create a new push session for its entity and request all rules needed to generate data update.
2. The server computes the rules for that instance. With the id provided by the instance, the server knows which group the entity is member of. The server knows also the global hierarchy and direction of the rules. With this information, the server knows all rules that implies pushing data for this instance.
   The server sends the session id and the rules (set of models and fields per model to synchronize).
3. The local instance creates the session with the number given by the server. It applies the rules in the right sequence. Each rule generates update by calculating all the local modifications, *i.e.* all the records where the date of the last modification is later than the date of the last synchronization task. One update is created per record and mentions the corresponding rule, exported

fields of the records are defined by the "included_fields" attribute of the rule. Each update record is saved in DB for that session and its state is set "To send".

Loop (4-6): Exits the loop when all update record store in the db are in the state "sent".

4. Local instance sends a packet of updates. The size of the packet is configured on the local instance. Each data update have the session_id given at step 2
   Server saves the updates of the packet.
5. Server sends an ACK.
6. Local instance marks updates contained in the packet as "sent" (to the synchronization server).
7. Local instance validates the push session, by sending the session_id given at step 2
   Server validates the updates of that session by setting their state on "validated" previously they were "pending" and sets a sequence number on the updates. The sequence number defines the order of future pulling session.
8. Server closes the session and notifies local instance.
9. Local instance deletes the session_id, the sync_date store at step 3, is set on all record that were synchronized. If those record have been modified between step 3 and step 9, those modification will be send during the next push session.

Resuming

In case of failure in the communication with the server, it's always possible to resume the process where it stopped as the local instance knows the last state of the session.
The "state" of the session is known accordingly of the update saved in the database:

- **No update to send in the database**: no session started.
  - **Resuming**: start a new session
- **Update record exist in the database, some with mark as "sent" other not**: the loop was interrupted.
  - **Resuming**: continue the loop with the first update not mark as "sent"
- **No update record in state to send but the session_id exist in the database:** the loop ended but the validation of the session never happened.
  - **Resuming**: send the validation again (operation 7).

The resume operation is available if a session exists on the local instance and it's not running.
If operation "4" or "5" fails, the packet will be sent again by the local instance when resuming. Some data update will be duplicated but since they are part of the same session and provide the same information, during pulling session, the result will be the same as if they are not duplicated.

If operation "7" fails, the validation of that operation will be sent again when resuming. If operation "8" fails, the local instance is not aware that the session has been successfully finished. The client will resume at step 7 and the server, since it has already validated updates, will answer again with the ACK. In the meantime the updates can already be pulled by other instances.

Cancellation

The synchronization engine does not include the cancel operation. However, the process ensures that it is technically feasible to develop such operation that cancels an ongoing push operation. This cancel operation would be available only before the session is validated. To cancel the push : client side, delete all update records created with the given session_id and then delete the session_id, this operation will set the system in the state "**no update to send in the database",** server side : delete all the update with the given session_id.

3.2.3.2. Pulling data/updates

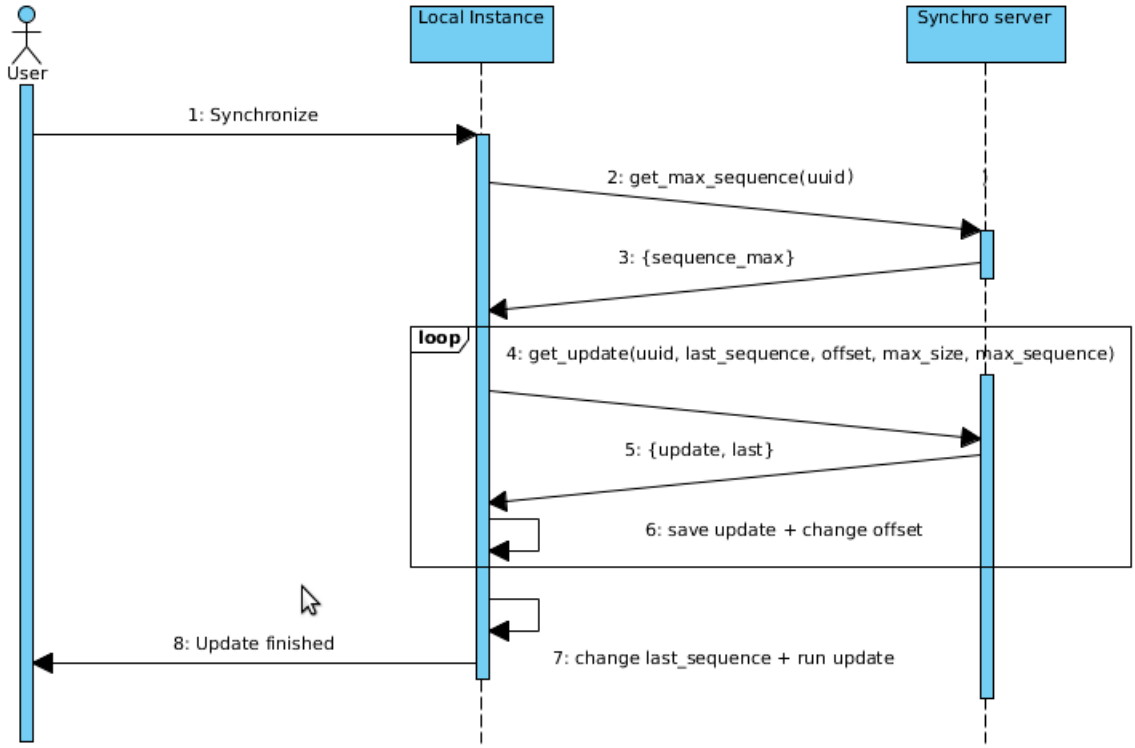
The data are pulled and applied in the order they have been pushed (confirmed) on the server (given by the sequence created at the end of the push process). The updates are already sorted as each update has already been sent in the right order.

Here is the summary of the normal process of pulling data.

1. The client asks the server a range of updates to pull.
2. Local instance retrieves packets of updates.
3. It applies the updates.

In the scope of this POC, an instance can run only one pull session at a time. The session must be ended before starting a new one. Nevertheless, it is technically possible to retrieve additional updates while not all previously retrieved updates have not been applied yet.

The following diagram describes the normal process:



1. The pull is triggered by the user (or a scheduled task) on the local instance.
2. Local instance asks the server a range of updates (there is one update per record to update) to pull.
3. Server returns the sequence of the last data update available for this instance.

   The server finds all updates in that range concerning the requesting local instance. It pre-processes data according to the rules applies forced values and includes fallback values.
   In case there is no more data to pull, the sequence max received is the same as last_sequence store in the instance's database.

   Loop (4-6): the loop ends when the server returns last set to true.
4. Local instance asks (next) packet of updates. The size of the packet is configured on the local instance.
5. Server sends a packet of update.
6. Local instance updates the offset. The offset is the number of update records received within a pull session.
7. When the local instance has received the last packet, it locally updates the sequence number of its last update and reset the offset and the max_sequence.

   Local instance applies/imports the updates sequence by sequence. In this step conflicts or errors may arise and manual user operation could be necessary

depending on the conflicts management policy defined. This is discussed later in this document

8. User is notified of the end of the process.

Resuming

In case of failure in the communication with the server, it is always possible to resume the process where it stopped as the local instance just asks for the next packet. The resume operation is available if a session exists on the local instance and it is not running.

The state of the pull operation is given by the last sequence and the offset.

- The offset and max_sequence are not equals to 0: the pull was interrupted in the middle of the loop

- Resuming: restart at step 2, but since the offset is not null, the package already sent are not sent anymore.

- **The offset and max_sequence are equals to 0** : the pull was finished or never started

- **Resuming**: restart at step 2

If operation "4" or "5" fails, the packet will be sent again to the local instance when resuming.

Cancellation

The synchronization engine does not include the cancel operation. However the process ensures that it is technically feasible to develop such operation that cancels an ongoing pull operation. This cancel operation would be available only before all packets have been received, *i.e.* before operation "7". To cancel the pull operation delete all the updates with a sequence number higher then the last_sequence number and reset the offset and max_sequence in the instance's database.

3.2.3.3 Rule and update routing

A rule defines which data transfer but also defines who push and who pull the data. A rule is defined for a group or a type of group. It will be communicated to instance that belong to this group. During the push operation, rules are sent to client but only if the entity belong to the same group as the rule. Sending the rule also depends on the direction of the rule, there is no point to send a rule with direction up if the entity has no parent in the group and symmetrically there is no point to send a rule with direction down if the entity has no child in the group.

The process is the same for pulling: each update is linked to an entity and a rule. Update will be sent only if the entity that requests this update belong to the same group as the generating rule. And the server will also respect the direction.

If a rule is defined on a type of group then this rule will act like it has been defined on all groups of this type. We can see this as a copy of the rule for all group of the same type. So even if the rule is defined for a type, entity of different group will not share data.

3.2.3.4 Remarks

Fallback values

Fallback values are sent by the server inside every update record but fallback value are there to replace value for relational field. The server will only send the xml_id of the fallback value. So we need a mechanism to ensure that fallback values are all present on every instance. For this, we impose that those records must be defined in the xml/yaml files that will be imported during deployment of a new instance or during upgrade of an existing instance. Those fallback values must be defined with an unique xml_id and be generic enough to fit all the cases required. If an update of a module adds a new required field (with the following type one2many, many2one, many2many), then a new fallback value might be required for this field and this value should be inserted during the upgrade.

Parallelism

In the scope of this POC, the push and pull operations cannot run in parallel, only one operation is allowed at a time and must end before starting the other one. However, it could be technically possible to run a push operation while all pulled data has not been imported yet. Push and pull operation are composed of two main steps (same for message): sending/receiving data and creating/importing update records.
To ensure data integrity the operation of creating/importing update records should never be run in parallel, but the communication with the server can be run in parallel. As other kind of parallelisms doesn't provide any benefits (speed, size), they are not covered here.

Transactionality of business operations

In OpenERP, a business operation can affect multiple models, for instance a sale order is made of a sale order record and sale order lines records. When synchronizing the business operation, it is important to export and import concerned tables at once to prevent for instance to synchronize a sale order without its lines. This is performed by the session mechanism, which ensures that all the updates of an export session are sent (pushed) before being electable for retrieval. In the same way, they are all retrieved (pulled) before being applied.

Computed fields

The standard modules of OpenERP contains computed fields like counters, for instance the stock level of a product. The computed fields are python functions calculating a

value based on other information, possibly from other records. OpenERP provides two mechanisms:

- the field is not stored and it's calculated when the record is read
- the field is stored in DB and an optional trigger is associated to recalculate its value based on other records. Those field are calculated when the record is modified or when records, which the field calculation depends on, are modified.

This ensures that computed fields calculation are always correct.

No clocks synchronization between instances

The described synchronization processes do not require server and instances to have an accurate clock. The server serves data according to a sequence instead of using a timestamp.

Each instance loads the data to export and to import according to its own clock. A record is pushed if the modification date is greater than the synchronization date. The modification date is set each time a record is created or modified. The synchronization date is set each time a record is validated by the push. The date is always the date of the instance itself. Note that OpenERP requires that the date on the instance is not modified to the past without analyzing the impact, this is also valid for the synchronization.

Suppression

The handling of the suppression of records is not part of the synchronization engine. It is not recommended to suppress records of synchronized models as this deletion won't be synchronized with other entities.

- **Which are the solution for suppression?**

We cannot simply override unlink method to set inactive records instead of deleting them for two reasons:

1. There is a on delete cascade attributes on some fields (for instance sales order lines in sales order) so that they are deleted with the parent directly at the database level (so the unlink method of sales order lines is not called).
2. Some models, that have no "active" field, have some reports (SQL views) that do not support "active" field and would thus need to be rewritten to be compatible with this new field.

This solution needs a lot of work.

But it is not a problem for a user to delete a synchronized record since the record will just appear again next time a change is pulled. A record will be deleted only if all instance that share this record delete it at the same time (and don't have any pending pull).

A solution to avoid unwanted deletion is to design access rule in a such way that synchronized model do not allow record suppression.

A solution to ensure deletion is to implement a deletion instruction update. Instead of containing modification or a new record, this update would contain an instruction to delete a record. Every entity pulling and running this update will delete the record. This solution could be implemented once the conflict detection works well.

3.2.4. Messages transfer

The transfer of messages is always done through the unique synchronization server. It is performed by two distinct operations:

- the push of messages that sends messages to the server for sharing with another entity.
- the pull of messages for this entity.

A message is an instruction created on a source entity to be executed by one target entity. Messages are generated according to message rules. A message rule contains all information needed to create a message when some record manipulation happens For instance, on a source entity if we create a purchase order with some target entity set as seller, then the system should create a message containing the instruction to create an appropriate sale order with the source entity set as buyer, and this message should only be interpreted on the target entity.
The information we need in a message rule are:

- the model name of the record on which the message creation is triggered.
- a domain (to select a subset of all records).
- the name of the method that will be called on the model in the target entity and also the values of the arguments of the call. Those values are calculated according to the values of the record that generates the message.
- the destination of the message which is also generated according to the values of the record that generates the message. A dedicated field (entity_name) allows to define which field is used.
- a group or type of group. The group or group type defines which group(s) of entity the rule applies to. Here, group just limits the number of instance who will generate a kind of message but destination can be an entity of a totally different group (so, destination is not defined by the group specified in the rule).

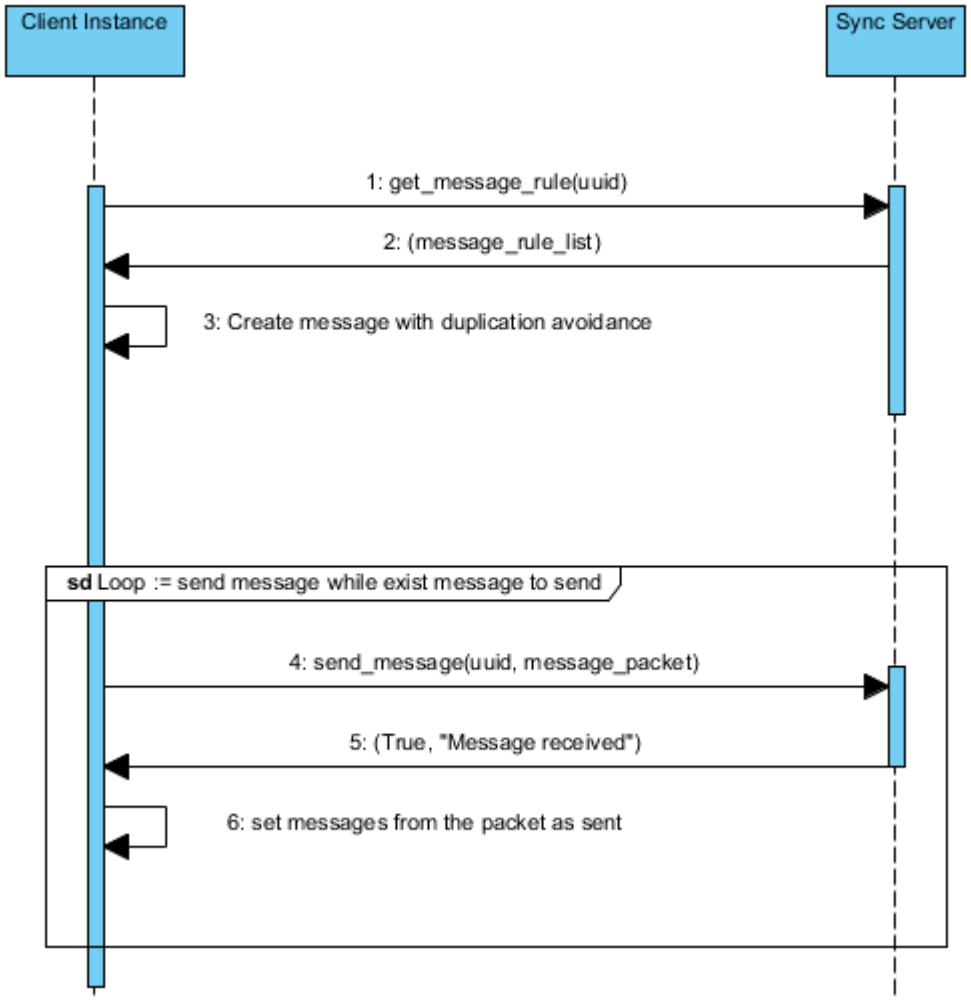
3.2.4.1. Pushing messages

Here is the summary of the normal process of pushing messages.

1. The client retrieves from the server the concerned rules in order to know what messages to generate and push.
2. Each new message is sent to the server.

An instance can run only one message push operation at a time.

The following diagram describes the normal process:



The message push is triggered by the user (or a scheduled task) on the local instance.

1. Local instance asks the server for message rules.
   Server collects message rules that involve that instance.
2. Server sends message rules.
3. Local instance generates messages one by one. Before generating a new message, it checks if the record is not already processed (that means a message has been generated by the same rule for the same record). It associates a unique id with the message. The unique id of a message is generated by the unique id of the generating record and by the id of the rule It allows the system to directly detect duplicate messages (one record can generate only one message per rules).

Loop (5-9): The loop stops when all the generated messages are sent.

4. Local instance sends the message to the server by packet.
   Server saves the message in its DB.
5. Server sends an acknowledgment for that message.
6. Local instance marks message as sent.

Resuming

In case of failure in the communication with the server, it is always possible to resume the process. It will restart from the beginning but will generate the next new messages.
If operation "4" failed, the local instance will resend that message to the server.
If operation "5" failed, the local instance will resend the message of operation "6" and the server will detect it is already received thanks to the unique id attached to the message.

Cancellation

The process can be cancelled at any time and resumed later. There is no such notion of cancellation for message, since messages are sent one by one. Messages are either sent or yet to be sent, there is no pending state.

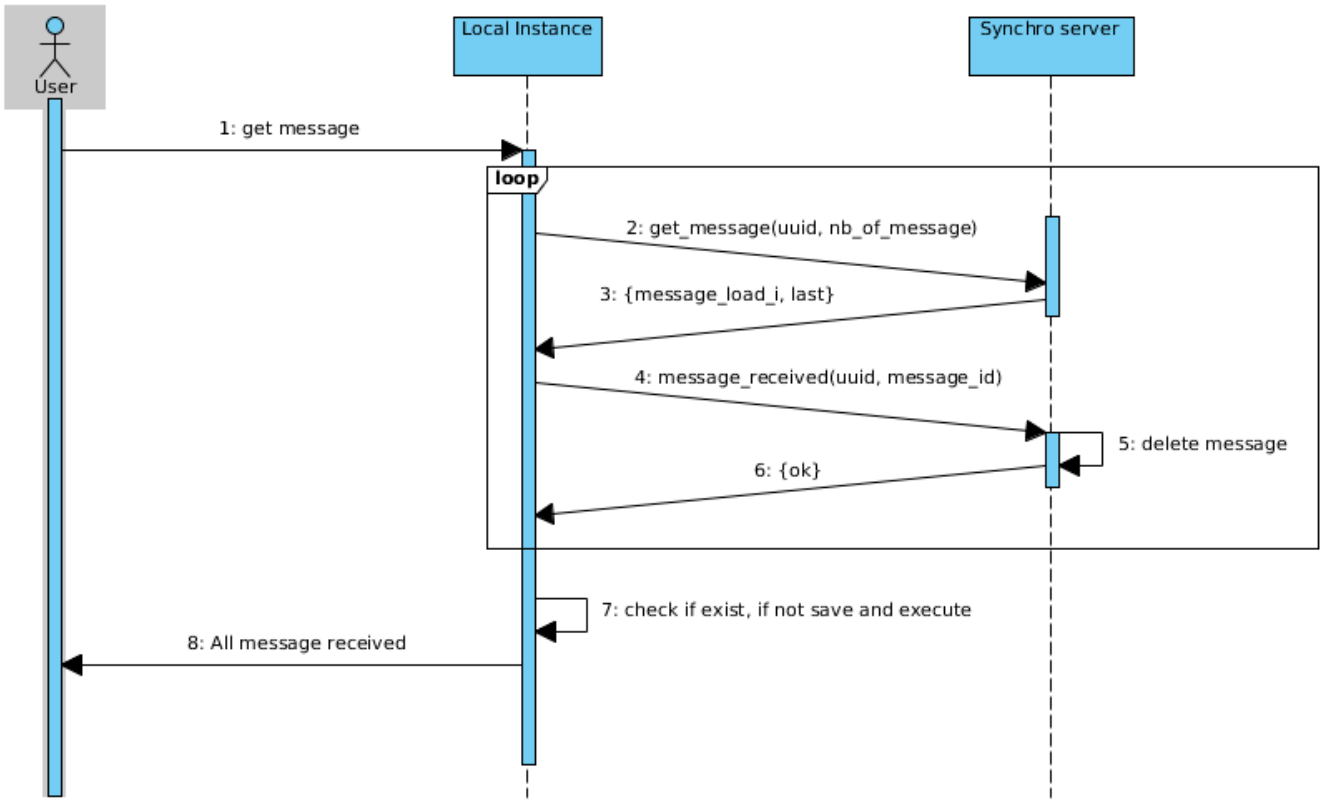
3.2.4.2. Pulling messages

Here is the summary of the normal process of pulling messages.

1. The client retrieves messages from the server.
2. It processes messages one after another.

An instance can run only one message pull operation at a time.

The following diagram describes the normal process:



1. The message pull is triggered by the user (or a scheduled task) on the local instance.

Loop(2-6) : The loop stops when the server answers last = True and the last ACK is received

2. Local instance asks for a packet of messages, with a given number of message.
3. Server sends a packet of messages
4. Local instance send the received validation message that contains the id of the received message.
5. Server sets the messages of the packets as "sent" but they are not deleted to detect duplicate message in case the source would send it again.
6. Server send an ACK to the client, the client knows that it can ask new messages.
7. Local instance processes the messages.
8. User is notified of the end of the process

Resuming

In case of failure in the communication with the server, it is always possible to resume the process where it stopped, as the local instance just asks for the next packet.
If operation "2", "3" or "6" failed, that packet is re-requested.

If operation "4" failed, that packet will be received twice and the local instance will detect it as duplicate.

Cancellation

The process can be cancelled at any time and resumed later. Their is no such notion of cancellation for message, since messages are sent one by one. Messages are either sent or yet to be sent, there is no pending state.

3.2.4.3. Remarks

Deferred acknowledgement

It could be possible that a message is pushed by an entity but that entity is not acknowledged due to communication failure (Pushing message, operation 9 fails). But the message sent could already be pulled by the destination entity and deactivated by the server. When the first entity resumes the push operation and re-sends the same message, the server detects this duplicated message (the message already exists in the database and is set as "processed") and acknowledges the pushing entity. This way, the pushing entity can continue to push new message (ACK is received) and the pulling entity won't receive twice the message.

Transactionality of business operations

A message must contain all the calls and information needed for a complete business operation. A message doesn't depend on other messages.

Specific case management

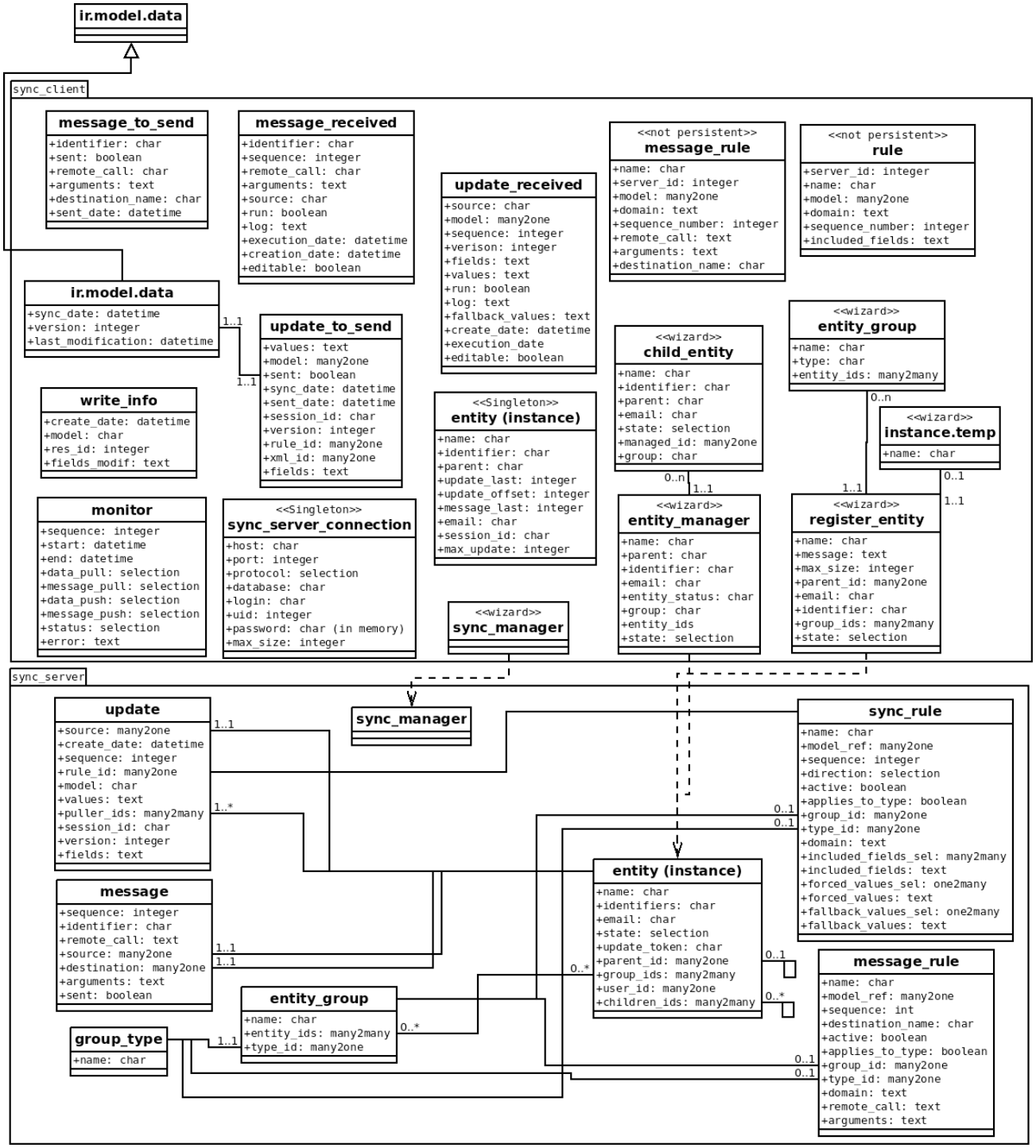
With message rules, we define a generic way to create message and to find the destination. The message is a remote call and argument of the call is the value of some field of the generating record. The destination is found by using the name and value of a field. Finally, the generation of the message is made when the rule is received.

All of these behaviors may need to be overridden in some case. We will offer method hooks for some model that need a more specific behavior (for instance, need to compute information for the argument of the call or need to compute information to define the destination).

3.2.5 Modules and Class diagram

The synchronisation is performed thanks to 3 modules. The sync_server module runs on the unique sync server, which has also an OpenERP instance installed. The sync_client runs on each OpenERP instances, sync_so runs on each OpenERP instances and contains specifics customization on sale order and picking in order to give an example of message.

This section partially presents the Class diagram providing the flows and functionalities described in the sections above and could evolve during the implementation phase of this POC.



3.2.5.1. Synchronization server

The synchronization server instance has a dedicated module "sync_server" that provides:

- List of OpenERP instances at the MSF entities (HQ, mission, field ...) structured as a tree hierarchy.
- Groups of instances (e.g. a particular mission or a section) and type of group
- Synchronization rules and messages specifying which model to synchronize and in which direction.
- List of updated records by each synchronization from an entity.
- List of message send by each instance.
- Sync_manager offering synchronization procedures to each OpenERP instances.

Here is the description of the class diagram. Following models are stored in DB (osv objects):

- The **entity** model defines an OpenERP instance. Each entity has a unique identifier generated at client side. The other attributes of an entity are the name, the identifier, an optional parent entity, the list of children and a list of groups it belongs to. Those values are set manually by the admin of the local entity.
- The **entity_group** model defines a group of entities. Attributes are the name, the type and the list of entities belonging to this group.
- The **group_type** model defines a type of groups, for instance *mission*, *section*, or *international*. Attribute is the name.
- The **update** model defines the list of records created/modified by an OpenERP instance (client) and already sent to the synchronization server. Attributes are the id, the source entity, and a dictionary containing the values to synchronize, the session_id and the sequence. The destination is computed at pulling operation according to the source and the generating rule. The list of entity that pulled the data update.
- The **message** model defines the list of messages send by instances to deliver to other instance. A message contains a sequence (the order of creation), an identifier to avoid duplication of message, the source instance and the destination instance, the method to call and the arguments of the call.
- The **sync_rule** model defines a synchronization rule. A synchronization rule has the following attributes:
    - a name;
    - a direction which is either *up*, *down* or *bidirectional*;
    - a model to select which table will be synchronized;
    - a domain, allowing one to filter the records to be synchronized based on criteria on fields values, select which row of the table to synchronized;
    - a sequence, defining the priority of the synchronization;

    - a list of fields *included_fields* containing the list of fields to include during the synchronization, to select which column of the table to synchronized;
    - a dictionary *forced_values* containing a mapping between field names and values. It sets or replace fields of the update in the pre-process of the push;

    - a dictionary *fallback_values* containing a mapping between field names and xml_id values. Such value will be used for relational fields. If the pulled data needs a record that does not exist in the destination database, the fallback id will be used. Of course a mechanism to ensures the presence of the fallback record in the destination database is needed. The fallback record must be inserted at the installation of the sync module (automatically in the xml/yaml file) along with a security rule that prevents to delete this fallback record.
    - a link to the model under concern in this rule;
    - a link to the group of entities and a link to a type of group of entities, but only one is taken into account, according to the value of the *applies2group* field.
    - Field *_sel are added for a better usability when defining the rule.
- The **message_rule** model defines a message rule.
    - *name*
    - *model* to define on which model the rule will apply
    - *domain* to select which records will generate a message among all records of a given model
    - the *remote_call* field specify just the name of the method to call (and on which model this method is defined
    - the *arguments* field specify how compute the argument of the call
    - the *destination_name* field specify how compute the destination of the message
    - *group_id*, *type_id* and *applies2group* are the same field as in sync_rule, they define which entity will generate message according to this rule

3.2.5.1.1 Rules management

Rule is an OpenERP model like any other, views are defined to allow the administrator to create, delete and modify any rules. During development phase, this interface will be useful to design rules. But once a rule is well defined, it is better to create an xml data for automatic and safe insertion of the rule during deployment. In production environment, rules management interface will allow, among others, to adapt and correct possible errors. But even with the rules management interface, modifying a rule still requires some technical skills.

3.2.5.2. OpenERP instances (client)

Each OpenERP client instance has a dedicated module "sync_client" that provides:

- Recording modification of any business records
- recording of the created and modified records
- Connection Manager : connection configuration
- Instance configuration and registration and synchronization wizards
- All sent (or to be send) update : use to resume the push process and for monitoring
- update received : use to resume the pull process in case all the updates of a session did not arrived yet and to monitor update errors
- all message already sent, to avoid sending duplicate messages
- message received to avoid executing twice the same message and for monitoring the error during message execution
- The client also stores temporarily the data synchronization rule and the message rule in order to generate data update and message.

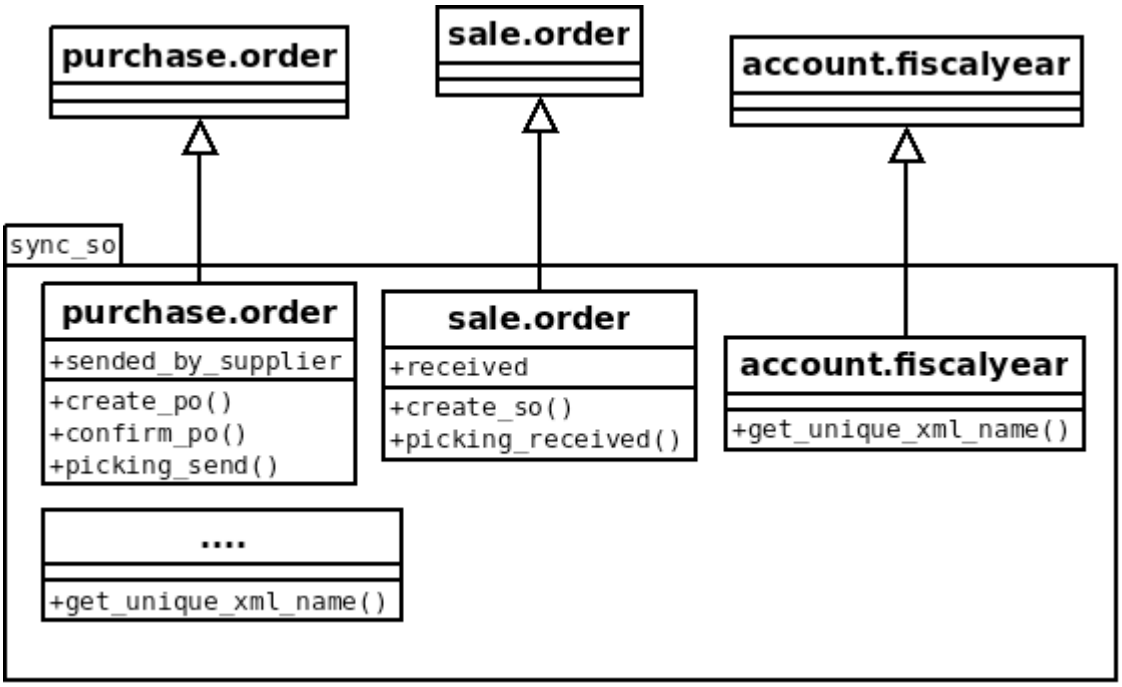
The module extends the **ir.model.data** model to add a *sync_date* field, containing the timestamp of the last synchronization of the corresponding record.

It creates a trigger after "create" (but also after the write in case the data was created before the installation of sync_client, and also when data are exported) , to link the new created record with a unique xml id (unique among all entity). To ensure that xml_id will be unique we build it like this: unique identifier of the entity + name of the model + DB id of the creating instance's database. When an update is pulled, this model will check if the xml_id already exists or not. If the xml_id exists then the related record is updated, if it does not exist then a new record is created related to an ir.model.data record that contains the xml_id received during the pulling operation.

To handle and detect conflicts another field has to be added to ir.model.data: the version of the record. This is discussed later in this document.
The module adds a new singleton model **entity** describing the current OpenERP instance: name, unique identifier, identifier of the parent and the id of the last update retrieved from the synchronization server.
The sync_manager implement the communication with the server, all the wizard needed for the interaction with the user.

3.2.5.2.1 Sync_so (client)

The client instance also has to install the sync_so module. This module contains new call that are used by message and also define the "get_unique_xml_name" for some model that need a particular case.



3.2.6. Synchronization scheduler at server side

There is no scheduler at server side. The connection is always initiated by the client as it is not always connected.

3.2.7. Exchange protocol

The exchange of data is performed with the OpenERP NET-RPC protocol. It consists in a python serialization using pickle and transmitted over TCP/IP. The port used is easily configurable. Data representation is very concise.
it is possible to compress data using GZip, as well as secure the transmission through the secured HTTPS protocol. Note that OpenERP already implements SSL over XML-RPC but not over NET-RPC. NET-RPC has been selected as the size of data is smaller than with XML-RPC.

3.2.8. Database storage of data and messages

3.2.8.1. Storage of data/updates

The local instance stores in DB update packets. The packets can be deleted from the local instance at the end of the process.

The server stores in DB updates for all entities and always keeps them all forever. There is no mechanism in the sync engine to limit the size of that table.

3.2.8.2. Storage of messages

The local instance stores in DB messages sent and received forever (at least their references) to avoid duplication of message and duplication of execution.
The server stores in DB messages for all entities and always keeps them all forever. There is no mechanism in the sync engine to limit the size of that table.

3.2.9. Size of the packets

A packet contains a fixed header and a list of updates or messages. An entity can configure the number of items inside the packets sent and received. As all items do not all have the same size, the packets will not have all the same size. As each packet has a fixed header, if small packets are used, more data will be transferred on the network, but if a packet is lost then the amount of lost data that has to be transferred again is smaller. If the quality of the connection is poor, smaller packets are better as they have more chance to arrive without timeout.

3.2.10. Conflict management rules

Definition of conflict: in data synchronization, there is a conflict when two (or more) different entities modify the same data between two synchronizations.

We describe some scenario to explain when there is conflict and when there is not, then we will analyze which tools we need to detect these conflicts.

We have three instances C1, C2, C3 with an already synchronized sale order (SO1), *i.e.* at the beginning of each scenario SO1 is the same everywhere.

Scenario 1 : local modification and pull
C1 : changes SO1 and push
C2 : changes SO1 and pull
=> conflict because C2 has changed SO1 locally and gets a remote change

Scenario 2 : two remote modifications
C1 : changes SO1 and push
C2 : changes SO1 and push
C3 : pull SO1
=> conflict because C3 gets two updates from different sources and those modifications were done simultaneously

Scenario 3 : two modifications by the same entity
C1 : changes SO1 and push
C1 : changes SO1 and push
C2 : pull

=> no conflict, C2 applies the two updates from C1

Scenario 4 : Two remote modifications
C1 : change SO1 and push
C2 : pull SO1
C2 : change SO1 and push
C3 : pull SO1
=> no conflict because C2 has integrated modifications of C1 before changing SO1

The difficulty is to detect the conflict at the right moment; the scenario 3 should not be detected as a conflict.

To detect conflict in scenario 1, we can use the synchronization date stored for each record and the modification date. The synchronization date is updated at each push and each pull, modification date is modified at each local change of the record. If modification date is greater than synchronization date then there is a conflict to solve.
With this solution, conflicts are only detected if there is a pull before a push

To detect correctly other conflicts, the best solution is to keep a version number of each record, increment this number when pushing and setting this number of the one given by the instance that pushes. Conflicts are detect when the version number of data pulled are not strictly greater than the local number.

Scenario 2 : two remote modifications
C1 : changes SO1 and push V2
C2 : changes SO1 and push V2
C3 : pull SO1
pull V2 then V2 => conflict because C3 get two update from different sources and those modifications were done simultaneously

Scenario 3 : Two modifications by the same entity
C1 : changes SO1 and push V2
C1 : changes SO1 and push V3
C2 : pull
pull V2 then V3 => ok

Scenario 4 : Two remote modifications, but no conflict
C1 : change SO1 and push V2
C2 : pull SO1 : pull V2
C2 : change SO1 and Push : V3
C3 : pull SO1 : pull V2 then V3 => ok

After detecting conflicts, a policy to solve them has to be define, general policy like last update pulled wins the conflict, or update from the higher entity in the hierarchy win. Another possibility is to let the user decides for each conflict. The best solution is a combination of those rules according to the model. Local instance need to store those rules. In a simple version, a rule has a name, the model for which the rule applies and the type (last, parent, manual).

The current implementation of the synchronization engine has not smart method to handle conflict. The last data update that comes will be the version saved.
However, the conflicts are detected and logged.

3.2.11. Monitoring system and error handling

The all process of synchronization is composed of four main steps: pulling data, pushing data, sending messages and receiving messages. Each step can generate errors despite that we try to avoid them by sorting the data in the right order.

Pulling data

- Error during communication. The operation can be resumed at any time and the session mechanism prevents from propagating partial data to other entities.
- Error during the execution of an update. Missing model in client instance (the destination of the update), or missing value of a required field in the update. The rules are not properly designed or adapted to the local version of OpenERP or a module is missing.
  The most common error when executing an update is a missing reference in many2one field. This error is because by data we forget to synchronize or because previous data update failed. Errors are logged as the update cannot be run. It's possible to visualize each failed updates, modify it and rerun it.
- Warning during the execute of an update. A value for a field is given in the update but not present in the model at the local instance. The rules are not properly designed or adapted to the local version of OpenERP. A warning is logged as it's not a blocking issue prevent update to be imported.

Pushing data

- Error during communication. The operation can be resumed at any time and the session mechanism prevents from propagating partial data to other entities.
- Error while creating an update package. Some fields are missing or some models do not exist in the local instance. Either rules are not well designed or do not fit the local version of OpenERP or some modules are missing on the local instance.

An error is logged and the local administrator can install missing module or contact rules designers to change them.

Sending messages

- Error during communication. The operation can be restarted.
- Error while creating a message. Same problem as pushing data can arise during message generation for the same reasons.

Receiving messages

- Error during communication. The operation can be restarted.
- Error while applying a message.
    - Some data are missing for the execution because the execution of the messages comes before the complete import of the required records.
    - The method to call does not exist : the message rule is wrong or the version of OpenERP does not match
    - The method call crashes : bug in the functional module

  It will be possible to visualize each failed messages and retry them or cancel them.

3.2.12. Logging mechanism

OpenERP provides a standard logging mechanism using the *logging* Python module. The fine tuning of the logged messages is not part of this PoC and could be performed in a next phase.
All logging message are create with the opener logger with the title 'sync.client' or 'sync.server' at debug level.

3.2.13. Handling of connection cuts

Micro connection cuts are handled by the TCP layer (best effort) which should be properly configured. Longer cuts, i.e. timeout or loss of NET-RPC call, can be resumed or restarted as already described in previous sections.
The current implementation of the client rpc library has a timeout of 3600 seconds and retry 10 times before the process is abort and has to be restart manually.
(Those value can be easily tune in rpc.py in sync_client module)

3.2.14. Good practices

3.2.14.1 Good practices to write rules

- Do no create a rule that ask a type of entity to push or pull data they do not have installed. For instance, do not ask an instance to push a sale order if the sales module should not be installed on the instance.

- If an instance sends a message to another instance, be sure that required data will be pushed by the sender and pulled by the receiver, or at least they share the required data. For instance, if an instance creates a purchase order that will create a sale order in another instance, ensure the product used in the sale order is shared by both instances.

3.2.14.2 Good practices for versioning

The synchronization server does not store any business record, the only constraint for the server is having compatible version of sync module with the version of the clients and vice versa.
But between two clients that exchange data, there is some rules to respect:
- Both clients should share the same model name: avoid renaming model between two versions that should work together
- All the required values should have a fallback value for all different versions. It's not a problem for a client to receive data that do not fit in any field but it's a bigger issue to have a record without some required field. When you design rules always give a fallback values in that case
- Avoid renaming fields between versions
- It is not a problem to add new fields between two versions, unless they are required or they do not have a fallback value define in the rule.

3.2.16 Additional comments not answered in the previous text

About version dependencies, as the sync server is basically just a store for synchronized record, only the client entities are "aware" of their own version. This simplifies the coexistence of different versions of OpenERP by backward compatibility: a client that is in a newer version just needs to support the calls serialized in the messages. The data can be synchronized without modification since we specify which fields to include in the synchronization rules and since we don't remove field in newer version. Although, any new version of OpenERP can definitely have a non-negligible impact on the synchronization of records whose data model and/or behavior have drastically changed.

As previously stated, we do not guarantee the possibility to connect to the sync server with 500 simultaneous connections. If the server is overloaded, the client will just retry later. A frontal proxy for load balancing is possible and easily deployable thanks to the stateless architecture of OpenERP, though we don't have tested this yet. This can come in a second phase of the project.

About authentication. The client entity is uniquely defined by its unique id. Synchronizing over SSL prevents from stealing the unique id by a man in the middle attack.

Automatic launch. This can be done in a second phase as it won't need to change anything in the existing implementation.

Online software updates

This project is about data synchronization and exchanging message between instance there is no mechanism to push software update from the synchro server to instance. This needs totally different mechanism and is out of the scope of this project.
A simple solution can be to download patches or binary deltas from a server (to be installed and maintained) and ask the local operator to apply them (unzip) and restart the software. We could automatize further but this needs its own analysis, though it is not too complicated. The workload could be in the order of days, but this value is not contractual.

Migration

Migrations consist of non-minor software updates, *i.e.* updates where the data model and/or behavior have changed and force an upgrade of the database as well. Migrations usually are a project on their own. This case is not the exception and is even more complex since multiple versions needs to run together. This needs its own analysis and is also more complicated than simple updates. The workload could be in the order of weeks per version and of days per instance, but this value is not contractual.

If the instance has a good Internet connection, then the migration of one database could be done as usual (service provided by OpenERP). This implies that the database is first anonymized, sent to us, migrated by us, sent back to the source and deanonymized. The anonymization process is automatic and optional. The software can just be downloaded from a server (to be installed and maintained) and installed manually by the local operator. Though, the synchronization modules add a complexity layer we are not used to in our usual migration service. The sync modules would need to be progressively adapted to support the different versions running in the park before migrations happen.

If the instance has not a good Internet connection or is prone to black out, then we suggest that the instance is never migrated (since it won't interact often, the migration is maybe not worth it). If this is really not possible, then the migration should be "tried" during a period when the connection is capable to sustain the process described above.