FNV/swua libswumagent API documentation

1. Summary

The steps to implement an agent are:

1. Extend the base class SwuAgent and override its life cycle and task callbacks
2. Instantiate the ECU-specific SwuAgent class
3. Instantiate the SwuAgentConfig class and set its properties
4. Instantiate the SwuAgentFramework and pass it to the ECU-specific SwuAgent as well as the SwuAgentConfig objects.

The SwuAgent, SwuAgentConfig, and SwuAgentFramework objects may be instantiated in their own process or may be incorporated as part of another executable. The executable that contains them must run with sufficient privilege to create/update the database as well as to download, install, and activate new software images and configuration data.

2. SwuAgent

The SwuAgent class is the base class for implementing the ECU-specific agent behavior. The methods of the SwuAgent are all callbacks that are invoked as the agent progresses through its life cycle and as SWUM tasks are run on the agent. The SwuAgent class methods provide empty implementations, allowing the ECU implementation to override them as needed. All callbacks are run either on the LCM thread (see below) for life cycle changes or individual task threads.

ECU-specific implementation:

(Note: some virtual methods are not listed because they do not need to be overriden. the default implementation fulfills the requirement.)

```
/**
 * Start a new transaction: create transaction folders, set permissions for download & install directories)
 * @param transaction The transaction
 */
virtual void onStartTransaction(SoftwareUpdateTransaction& transaction);

/**
 * Download the file(s) for the update of a transaction
 * @param transaction The transaction
 */
virtual void onDownload(SoftwareUpdateTransaction& transaction);
```

```
    /**
     * Install the update for a transaction
                1. check for destination
                2. validate manifest
                3. install
     * @param transaction The transaction
     */
    virtual void onInstall(SoftwareUpdateTransaction& transaction);

    /**
     * Activate the update for a transaction (switch partitions)
     * @param transaction The transaction
     */
    virtual void onActivate(SoftwareUpdateTransaction& transaction);

    /**
     * Reboot the ECU
     */
    virtual void onReboot();

    /**
     * Inform the agent that a reboot is occurred on the ECU.
     * The agent could do necessary checkup and report reboot task complete result
     */
    virtual void onPostReboot();

   /**
     * Cancel the activation for a transaction (like swap partitions)
     * @param transaction The transaction
     */
    virtual void cancelActivation(SoftwareUpdateTransaction& transaction);

    /**
     * End a transaction. Once a transaction is ended, it will be removed from the DB
     * @param transaction The transaction
     */
    virtual void onEndTransaction(SoftwareUpdateTransaction& transaction);

    /**
     * Upload resource from to ECG via sftp client
     * @param taskUuid The task UUID
     * @param sourceUrl The source url of resource to be uploaded
     * @param targetUrl The target url of resource to be uploaded to
     */
    virtual void onUpload(const std::string& taskUuid, const std::string& sourceUrl, const std::string&
targetUrl);

    /**
     * Retrieve the VIL file from ECG after OTA manifest is handled from USB, and save the VIL to the USB via
sftp
     * @param taskUuid The task UUID
     * @param vilReportUrl The url of VIL report to be downloaded
     */
    virtual void onGetVIL(const std::string& taskUuid, const std::string& vilReportUrl);

    /**
     * A callback to notify that a sequence is failed, agent could use this chance for necessary cleanup work
     */
    virtual void onSequenceEnd(const std::string& sequenceId);
```

2.1. SwuAgentConfig

The SwuAgentConfig class contains all of the customization parameters used to configure the agent, such as the node ID or database filename.

```
g_pAgentConfig = std::make_shared<fnv::swum::SwuAgentConfig>();
g_pAgentConfig->nodeId = 0x7D0;      // same as SYNC, this is used by ECG swumd to distinguish different ECUs
g_pAgentConfig->databaseFilename = fnv::swum::SWUAGENT_BASE_PATH + std::string(fnv::swum::SWUAGENT_DATABASE);
// database used to store transaction information
SoaConnectionOptionsIpc::SharedPtr soaOptions = SoaConnectionOptionsIpc::createConnectionOptionsIpc();
g_pSoaUtil = std::make_shared<fnv::swum::GatewaySoaUtil>(soaOptions); //set up soa
g_pAgentConfig->soaUtil = g_pSoaUtil;
g_pAgentConfig->sftpKeyFilename = ""; //The sftp key is used for uploading files to the ECG via sftp client,
will be sent to ECG for authentication
```

2.2. SwuAgentFramework

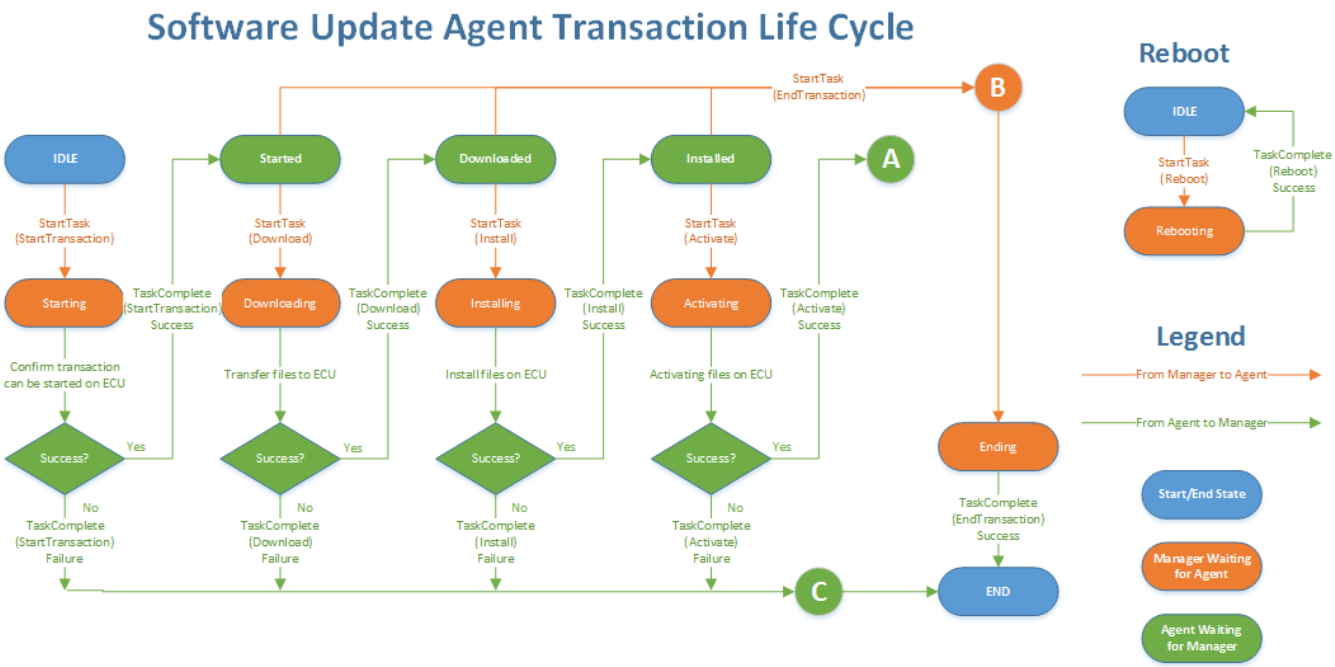
The SwuAgentFramework contains the core agent functionality for managing the life cycle, communicating with the SWUM, and running software update tasks.

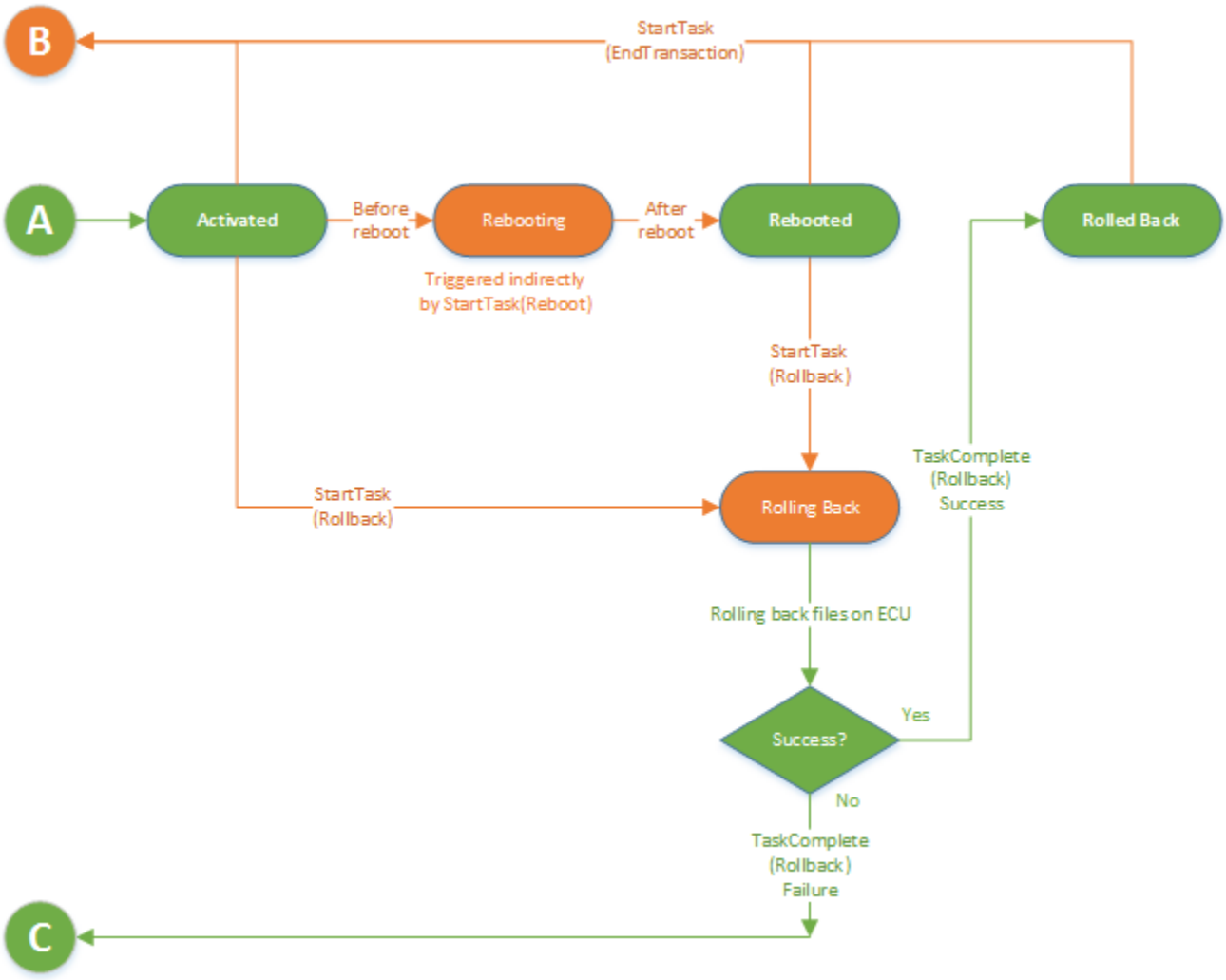
g_pAgentFramework = std::make_shared<fnv::swum::SwuAgentFramework>(g_pSyncAgent, g_pAgentConfig);

g_pSyncAgent->setAgentFrameWork(g_pAgentFramework);

3. Transaction Life Cycle

3.1. State Machine





3.2. Task Management RPCs

3.2.1. StartTask

The manager directs the agent to begin an asynchronous task by making a StartTask RPC to the agent. The StartTaskRequest includes the task type as well as type-specific task parameters.

The agent responds to a StartTask RPC by sending a StartTaskResponse. The response has a parameter that indicates success or failure as well as the reason for the failure.

If the agent is in the correct state and is able to successfully start the task, the agent moves to the next state and allocates a new random task ID for the task. The agent responds by sending a successful StartTaskResponse that includes the task ID. The agent then begins to execute the task asynchronously.

If the agent is unable to begin the task (such as being in the wrong state for the type of task, referencing an invalid transaction ID, or some other such error), it responds with a failure StartTaskResponse that indicates the reason for the failure. The agent does not change state.

3.2.2. TaskComplete

When the agent completes a task, it indicates its status by making a TaskComplete RPC to the manager. The TaskCompleteRequest includes the task type, task ID, node ID, and other task-specific parameters as well as a success/failure indicator.

The manager responds to a TaskComplete RPC by sending a TaskCompleteResponse. The response has a parameter that indicates success or failure of receipt of the request.

If the agent is unable to contact the manager, the agent will queue the TaskComplete RPC and repeatedly attempt to contact the manager until either confirmed success or failure.

In the event of a failure for a task, the agent will reset the state of the agent back to where it was when the task began. If the task pertains to a transaction, the agent will cancel the transaction and restore its state (as closely as possible) to what it was when the transaction began (cleaning up database records, temporary files, system configuration, etc).

Once the RPC is complete and confirmed to be received by the manager, the agent changes to the next state.

All fields of a transaction and utilities retated to a transaction (such as get or set a field of a transaction) can be found in software_update_transaction.hpp.

3.3. Transaction States

3.3.1. Idle

In the Idle state, the agent is ready to begin a transaction.

When the agent receives a Start Transaction StartTask RPC, the agent prepares to process a transaction and moves to the Starting state.

3.3.2. Starting

In the Starting state, the agent is preparing to begin a transaction. The agent creates a transaction object in its database.

When the agent finishes its preparations, it sends a StartTransaction TaskComplete RPC to the manager and moves to the Started state. The TaskComplete RPC includes information about the agent's properties, such as the agent's download type (direct or indirect), encryption keys, as well as other information about the service.

If the task fails, the agent removes the transaction object and sends a failed StartTransaction TaskComplete RPC to the manager. The agent them moves to the End state.

3.3.3. Started

In the Started state, the agent is ready to start downloading the files that are needed to process the transaction.

When the agent receives a Download StartTask RPC, the agent creates a transaction using the transaction ID provided in the request. The Download request also includes metadata for each of the files that the agent is to download. The agent creates a transaction object in its database, sends its response to the manager, then moves to the Downloading state.

If the agent receives an EndTransaction StartTask RPC, the agent moves to the Ending state.

3.3.4. Downloading

In the Downloading state, the agent is transferring the installation files from the location(s) given in the Download StartTask request.

For direct download devices (such as SYNC), there will be a single file to download in VBF format. The agent will download the VBF and split it into its respective files (one per data section). The agent will update its metadata with the split file data.

For indirect download devices (such as ECG and TCU), the transaction info received in download request will include the FileInfo information for each of the files downloaded by the manager. The manager will have already downloaded the VBF and split it into its respective files. The agent will add FileInfo entry in the transaction for each file extracted from the downloaded VBF.

When the agent is finished downloading, it sends a Download TaskComplete RPC to the manager and moves to the Downloaded state.

If the task fails, the agent removes all temporary files from the file system, all of its transaction data from the database, and sends a failed Download TaskComplete RPC to the manager. The agent then moves to the End state.

3.3.5. Downloaded

In the Downloaded state, the files to be installed for a given transaction are stored on the agent ECU.

If the agent receives an Install StartTask RPC, the agent begins the process of installation. It sends a successful response to the manager and moves to the Installing state.

If the agent receives an EndTransaction StartTask RPC, the agent moves to the Ending state.

3.3.6. Installing

In the Installing state, the agent is installing the downloaded files into the ECU.

First, the agent locates two files in the transaction: the manifest and the manifest signature. The agent calculates a signature for the manifest and compares it to the manifest signature. If they match, the transaction can proceed. If not, the task fails (see below).

If the manifest has the correct signature, the agent reads the manifest and populates the transaction database with the file metadata read from the manifest.

The agent then computes the size and hash for each file and compares those values to those in the manifest. If all sizes and hashes match, the transaction can proceed. If not, the task fails.

If all files have the correct size and hash, the agent installs each in a manner appropriate to the file type. If installation is successful for all files, the transaction can proceed. If not, the task fails.

Once all files have been installed successfully, the agent deletes the files from the ECU but preserves their metadata in the transaction object. The agent the sends a successful Install TaskComplete RPC to the manager and moves to the Installed state.

If the task fails, the agent sends a failed Install TaskComplete RPC to the manager. The agent removes all transaction data and temporary files and moves to the End state.

3.3.7. Installed

In the Installed state, the agent has installed all of the files in the transaction and is waiting for activation.

If the agent receives an Activate StartTask RPC, the agent begins the process of activation. It sends a successful response to the manager and moves to the Activating state.

If the agent receives an EndTransaction StartTask RPC, the agent moves to the Ending state.

3.3.8. Activating

In the Activating state, the agent prepares the ECU to use the newly installed software.

Activation may or may not require a reboot (see the Reboot task below). Rebooting the ECU does not cause the transaction to change state.

Once all of the files have been triggered for activation, the agent sends a successful Activate TaskComplete RPC to the manager and moves to the Activated state.

If the task fails, the agent undoes any activation triggers, removes all temporary files and transaction data, sends a failed Activate TaskComplete RPC to the manager, and moves to the End state.

3.3.9. Activated

In the Activated state, the agent has activated all of the files in the transaction. The ECU may or may not have been rebooted. The agent is waiting for the manager to determine if the activation was successful or not and to take action accordingly.

If the agent receives an EndTransaction StartTask RPC, the agent moves to the Ending state.

If the agent receives a Reboot StartTask RPC and the transaction's requires reboot flag is set, then before requesting a reboot, the agent will move the transaction to the Rebooting state. Once all transactions have been moved, to the Rebooting state, the agent will request a reboot.

If the agent receives a Rollback StartTask RPC, the agent begins the process of reverting to the configuration of the ECU that existed prior to installation. The agent moves the transaction to the Rolling Back state.

3.3.10. Rebooting

In the Rebooting state, the transaction requires a reboot, and the agent has requested a reboot.

If a transaction's requires reboot flag is set, then after rebooting the agent will move the transaction to the Rebooted state.

3.3.11. Rebooted

In the Rebooted stated, the transaction requires a reboot, the agent has requested a reboot, and the reboot has occurred. Post-reboot checks occur and are reported back to the manager in the Reboot TaskComplete RPC.

If the agent receives an EndTransaction StartTask RPC, the agent moves to the Ending state.

3.3.12. Rolling Back

It's not necessary to override this method. The framework’s default implementation is good enough. Instead, they should implement “void cancelActivation (SoftwareUpdateTransaction& transaction)” method, which will be called by the onRollback() method

In the Rolling Back state, the agent undoes the installation and activation of the files in the transaction.

Once all of the files have been rolled back, the agent sends a successful Rollback TaskComplete RPC to the manager and moves to the Rolled Back state.

If the agent is unable to successfully roll back all of the files in the transaction, the agent removes all temporary files and transaction data, and sends a failed Rollback TaskComplete RPC to the manager. The agent then moves to the End state.

3.3.13. Rolled Back

In the Rolled Back state, the agent has successfully undone the installation and activation of all of the files in the transaction.

If the agent receives an EndTransaction StartTask RPC, the agent moves to the Ending state.

3.3.14. Ending

In the Ending state, the agent ends its processing of a transaction. It removes any temporary files created to support the transaction, takes any end-of-transaction steps appropriate to the transaction (which may be different depending upon state - Started, Downloaded, and Installed will cancel the transaction, whereas Activated, Rebooted, and Rolled Back are proper end states), and removes all transaction data from its database.

Once all cleanup tasks have completed, the agent sends a successful TaskComplete RPC to the manager. The agent then moves back to the Start state.

If there is a failure during clean, the agent completes as much cleanup as possible, then sends a failed TaskComplete RPC to the manager. The agent then moves to the End state.

3.3.15. End

In the End state, the agent has no remaining work related to the transaction. All temporary files, database objects, and persistent state records are removed. All tasks have completed and reported their status. The transaction ID is unallocated and may be safely reused.

Notes: A SWUM campaign might contain several groups. In a single group, all transaction will be either succeed, or fail together. If any transaction in a group fails, the whole group will be failed. The onSequneceEnd() method will be called when a group is failed.

4. Framework Interface

The two usb update functions should be called within usb callbacks.

onTaskComplete should be called for every task either failure or success (note this is NOT a callback function).

```
/**
 * Reports to the framework that an manifest file from USB Update was received. This notification will be
 * sent to SWUM for further processing.
 * @param fileName Full path name to the manifest file received over USB
 * @return true if notification successfully delivered, false if not
 */
bool usbUpdateReceived(const std::string& fileName);

/**
 * Reports to the framework that an manifest file from USB Update was removed.
 * @param fileName Full path name to the manifest file removed over USB
 * @return true if notification successfully delivered, false if not
 */
bool usbUpdateRemoved(const std::string& fileName);

/**
 * Report task complete result
 * @param result the result of the task complete details
 */
void onTaskComplete(SwuAgentTaskResult& result);

/**
 * Report task complete result for a transaction
 * @param transaction the transaction which the task was executed
 * @param result the result of the task complete details
 */
void onTaskComplete(SoftwareUpdateTransaction& transaction, SwuAgentTaskResult& result);
```

5. Database Manager

DataManager is used to store the current state of the agent and the software update process.

The DataManager is used from all the components of SWUM to store current state of the agent and the software update process. It needs to store the following information:

- save the transaction id for the current active transaction (createTransaction(...))
    - there could be two active transactions at the same time possibly

- save the information from the commands received from SWUM (call createTaskRequest(...))
- save download and install progress if applicable (call void setUpdateProgress(int progress), then updateTransaction(...))

The DataManager is used to determine after each reboot where in the progress of the software update the SWUA is, so that the right component can be started to pick-up where it left off.

Please see swu_database_manager.hpp to perform those tasks.

Note: most of the utilities from this Data Manager are used by framework, the DuerosAgend do not need to make explicit calls. updateTransaction(...) is one of the functions that could be called explicitly and frequently for many use cases. Such as, when an agent restarted due to unexpected reasons. The agent can check the database to see if there are any unfinished tasks, for that the agent need to save the progress and status. However, if Dueros wishes to restart the whole software update lifecycle (instead of picking up where it left off), then DuerosAgent may not need to use the Database to save the status. All in all, it depends on what Dueros decides to implement.

6. HMI Interface

Option 1: pass through swuagent

Create a hmi handler within the agent and communicates over ipc with android hmi, using HMI message protocol. The handler's job is to notify the android hmi to update screens etc.

Option 2: HMI talks directly to ECG (as a soa client through soagateway)

OTA_HMI protocol

Generic OTA-HMI IPC

7. Download & Install Manager Implementation Suggestions

The Download and Install Manager should communicate (mq or soa, or some ipc mechanism) with the agent, and the agent will invoke its implemention of the callbacks, if download and install tasks are done asynchronously.

Suggestion: start with synchronous download and update to avoid callback implementations.

8. USB Manager Interface

China team will need to implement the mechanism for monitoring usb insertions removals, and notifying usb manager. The usb manager will then do some checks (such as if there is a package on the usb), and notify the agent, the agent and framework will handle the rest of the upload.

Usb update flow on sync

(pps usb handler) usb manager SwuAgent (maybe can combine usb handler and manager to one entity, up to the China Team to decide)

() should be implemented by China Team.

SwuAgent ECG (have usb available)

ECG SwuAgent (send credentials)

authentication...

SwuAgent ECG (onUpload should be invoked, uploading)

after finish uploading

ECG SwuAgent(save VIL file, a file needs to be saved somewhere at the end of usb update cycle)

There is some code can be reused/ported in usb manager. Or a usb manager header can be published for China to implement.

Notes: when communicating components that are separate processes outside of the SwuAgent (cpm, osinstaller), ipc mechanism (soa, ipc-lite, mq, or some kind of client-server implementation) shouldto be used. In the DuerOs case, hmi, download manager and install agent have to use ipc mechanisms to communicate with the DuerosAgent. If they choose to implement everything in a synchronous fashion, callbacks are not needed.