NMF Autonomy experiment (TN)

MPSS

Exported on Jul 08, 2020

Table of Contents

1 Introduction

This work implements an on-board autonomy NanoSat MO Framework (NMF) App using CCSDS-MP services (Planning Request service, Plan Information Management service, Plan Execution Control service), COM services (Archive service, Event service) and MC services (Action service). The Autonomy scenarios, planner, image classifiers, and the visibility window calculation are provided by the APSI team. The Autonomy App implements two on-board autonomy scenarios, and they can be executed concurrently in the Autonomy App:

Autonomy Scenario 1 plans for image capture using on-board camera. Re-planning is attempted until a clear image is taken or timeout.

Autonomy Scenario 2 scouts an area for interesting targets (e.g. volcano, reef). If an image is classified as interesting then "survey" images are taken. The image capture uses the system developed in Scenario 1.

The target deployment is on the OPS-SAT mission.

1.1 Tasks performed / features

MP core source code: The NMF implementation of a simplified MP Plan Execution Control Service that executes timelines. The execution is performed using MC Actions.

MP core source code: MP data structures update to model version "K".

Core source code: Clock service addition to Platform services. This allows the Platform services simulator to manually fast-forward simulated system time, which allows realistic Autonomy App demonstration, since the planned activities are spaced hours or days apart.

Application source code: Autonomy App Scenario 1 NMF design, implementation, unit and integration tests.

Application source code: Autonomy App Scenario 2 NMF design, implementation, unit and integration tests.

Application source code: APSI planner and classifiers (ESA licence only) used in Autonomy App, and added to source control.

Consumer Test Tool: MP services support. Allows submitting requests and monitoring request, activity, plan status live feedback.

Consumer Test Tool: Capability in MP panels to refresh published events from COM Archive starting from a specified data. This allows using the CTT as a monitoring center in the operational phase of the experiment.

Documentation: Autonomy App demonstration steps using SDK and OPS-SAT-like environment, this document and Sphinx export.

1.2 Project Goals

The work is an intersection of demonstrating on-board autonomy using MP services on one hand and APSI planning, re-planning an image classification on the other hand.

The goals of the Autonomy experiment are listed below. They are presented as requirements in order to evaluate design options and summarize findings at the end.

REQ-1 MP services and data model demonstration. The Autonomy experiment models a closed-loop on-board planning/execution system. The system components interact with each other using MP and MC services. The Autonomy NMF App shall demonstrate MP services capabilities, such as expressing the problem using the MP data model, controlling the system using MP interfaces, and monitoring request/activity life cycles.

REQ-2 On-board autonomy demonstration using MP services. Demonstrate the planning-replanning cycle using planned activity execution monitoring. The results are demonstrated using the Consumer Test Tool and the NMF SDK simulator.

REQ-3 APSI components demonstration. Demonstrate the APSI planner and Classification algorithms.

REQ-4 MP Service NMF implementations demonstration. The MP Planning Request, Plan Information Management, Plan Distribution, and Plan Edit services have been prototyped in NMF in the predecessor project. The services were prototyped such that the App-specific service operation over-rides can be implemented in adapters. This has been demonstrated using tests in the previous projects. Now demonstrate the NMF MP services by implementing the Autonomy scenario.

1.3 Diagram notation

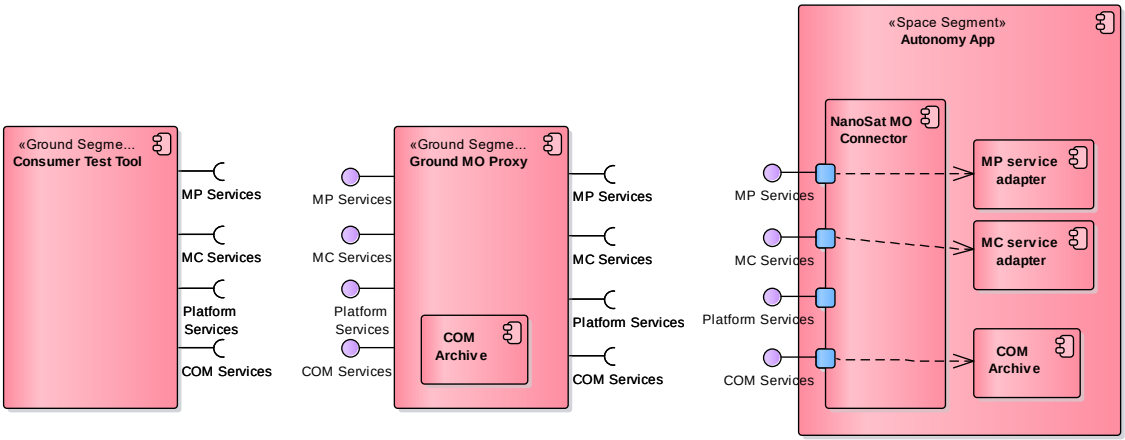
The UML sequence diagrams are for informative purposes. Events are shown using dashed lines. Method names describe activities and are not meant to correspond to code.

1.4 Experiment setup: App and Consumer Test Tool

The figure below gives an overview of the Autonomy app setup. Like any other NMF App it provides services, with app-specific over-rides implemented in service adapters. MC adapter for example implements a TakeImage action, MP adapter implements TakeImage activity.

The Autonomy App is controlled using the Consumer Test Tool (CTT), which is part of NMF SDK. The CTT sends the initial request and monitors the experiment events. In a development/test environment the CTT interacts with the Autonomy App directly using the MO interfaces. In an OPS-SAT-like environment the MO communication is buffered using the Ground MO Proxy, that acts as a protocol bridge (maltcp → malspp). The Ground MO Proxy also synchronizes the COM Archive when contact with satellite is re-established during a pass.

The request, activity, and plan statuses are transmitted using MO publish/subscribe interaction pattern over the MP service interfaces, when there is a direct contact with the satellite. The CTT has a capability of refreshing events from (the Ground MO Proxy) COM Archive, in order to retrieve the events that were published by the App when satellite is not visible (offline). This assumes the Ground MO Proxy COM Archive is synchronized with the on-board COM Archive (there are CTT instruction for that). The events monitored by the CTT are also published to dedicated monitoring logs by the App on-board, which can be retrieved by flight control team.



2 Abbreviations

APSI Advanced Planning and Scheduling Initiative

CLI Command Line Interface

CTT Consumer Test Tool

CCSDS Consultative Commitee of Space Data Systems

DDL Domain Description Language

MAL Message Abstraction Layer

MC Monitoring & Control

MO Mission Operations

MP Mission Planning

NMF NanoSat MO Framework

PDL Problem Description Language

PDS Plan Distribution Service

PEC Plan Execution Control Service

PED Plan Edit Service

PIM Plan Information Management Service

PRS Planning Request Service

SDK Software Development Kit

TLE Two Line Element

TN Technical Note

WG Working Group

3 Terms

Action - refers to MC Action service action.

Activity - refers to MP activity entity. Activity is considered a higher level construct than action.

APSI jar - ops-sat-plan-generation.jar, provided by APSI team, contains APSI framework, planner, image classifiers, visibility window routine.

Autonomy App - refers to the NMF app in <project_root>/sdk/examples/space/mp-autonomy-demo

<project_root> - location of the "nanosat-mo-framework" repo clone

Request - refers to MP PlanningRequest.

4 References

4.1 MO references

Mission Operations Message Abstraction Layer / CCSDS 521.0-B-2 / 03/2013
Mission Operations Common Object Model / CCSDS 521.1-B-1 / 02/2014

4.2 NMF references

https://nanosat-mo-framework.readthedocs.io/en/latest/

4.3 NMF-MP references

NMF-MP is the MP service specification that is implemented in NMF. In the "nanosat-mo-framework (mission-planning branch) navigate to <project_root>/core/mo-services-apis/mp-nmf/bin/target/xml/ServiceDefMP-nmf.xml (uses MP model 'K')

4.4 CCSDS-MP references

The CCSDS-MP specification is Work in Progress.

4.5 APSI references

On-Board Autonomy Operations for OPS-SAT Experiment / Simone Fratini, Nicola Policella, and Ricardo Silva

Apsi Timeline Representation Framework v. 3.0 / TECHNICAL NOTE / Apsi Modeling Languages / APSI-TRF3-MDL / 31/12/2014

APSI-BASED DELIBERATION IN GOAL ORIENTED AUTONOMOUS CONTROLLERS / Simone Fratini, Amedeo Cesta, Riccardo De Benedictis, Andrea Orlandini and Riccardo Rasconi

APSI Framework 3.0 / Simone Fratini, Nicola Policella / 11/10/2016

From Scheduling to Planning with Timelines: A history of successful applications in Space (Part 2) / Simone Fratini / 04/06/2013

Deploying Interactive Mission Planning Tools – Experiences and Lessons Learned – / Amedeo Cesta, Gabriella Cortellessa, Simone Fratini, Angelo Oddi, and Giulio Bernardi / Jc15-8-5107 / 25.05.2011

5 Dependencies

The Autonomy App uses APSI planner and image classifiers (ops-sat-plan-generation.jar), which is available for ESA users.

6 Installation

The code runs using Java 8.

Clone the "nanosat-mo-framework" repo from ESA gitlab or github. There are MO framework dependencies (CCSDS_MO_XML, CCSDS_MO_APIS) which may need to be installed to your local maven repo (clone the repos and run "mvn install"):

https://github.com/esa/CCSDS_MO_XML.git
https://github.com/esa/CCSDS_MO_APIS.git

Checkout "mission-planning" branch and build it. APSI jar needs to be installed before the build. See the README.md file in <project_root>/sdk/examples/space/autonomy/lib.

Note! The APSI jar needs to be installed every time it is updated. See the README file in <project_root>/sdk/examples/space/autonomy/lib.

7 Installation cross-check / tests

Run tests:

```
$ cd <project_root>/core/mo-services-impl/ccsds-mp-impl
$ mvn -Dmaven.test.skip=false test
```

```
$ cd <project_root>/sdk/examples/space/autonomy/
$ mvn -Dmaven.test.skip=false test
```

```
$ cd <project_root>/sdk/examples/space/mp-autonomy-demo
$ mvn -Dmaven.test.skip=false test
```

8 Configuration

This section explains the configuration and input files to the Autonomy App. The files in configurable folders allow fine-tuning the app after deployment, and specify folders for the artifacts generated by the app.

8.1 provider.properties

This is a standard file in NMF Apps.

Note. In SDK this file must be configured after compilation.

Property	Description
esa.mo.com.impl.provider.ArchiveManager.droptable=true	Resets COM Archive on App start. Default value is false.
esa.mo.nmf.app.systemTimeProvidedByPlatformClockService=true	System time provided by Platform.Clock service. Used with Platform services simulator. Default is false.
esa.mo.nmf.centralDirectoryURI=maltcp://10.0.2.15:1024/nanosat-mo-supervisor-Directory	Directory service provider, in case the Autonomy App is started from CLI. Optional.

8.2 application.properties

This file is specific to Autonomy App. The table lists the properties and their default values:

Property	Description
autonomy.experiment.folder=.	absolute path to autonomy experiment logs NB! folder must exist and have write permissions. Fallback to applicaton root
autonomy.app.images=${autonomy.experiment.folder}/images	images produced by camera acquisition
autonomy.app.planning=${autonomy.experiment.folder}/planning	APSI PDL and plan files
autonomy.app.monitoring=${autonomy.experiment.folder}/monitoring	Published MP request, activity, plan version, plan status events, corresponds 1:1 to similar displays in CTT
autonomy.app.plan.resume=false	If true, load plan from Archive. Use this when the App closes/dies, plan not finished,

Property	Description
	but the experiment needs to be resumed.
camera.maxangle=15d	Camera acquisition max. angle
adcs.setattitude.duration=30	In seconds, same as DDL worst case Locking + Locked
vw.planning.max.number.per.coordinate=10	Cap the number of VWs to consider, impacts performance
orekit.navigation.folder=orekit	Orekit navigation folder
apsi.ddl.file=apsi/OPS_SAT_Domain_edt.ddl	APSI DDL file
classifier.cloud.model = classifier-models/cloud_model.h5	cloud classifier model
classifier.interesting.model = classifier-models/interesting_model.h5	interest classifier model
apsi.temporal.start=2020-01-01T00:00:00.000Z	corresponds to DDL, this is the PlanVersion start time, APSI temporal start == PlanVersion start == PDL release
apsi.temporal.end=2020-12-31T23:59:59.000Z	corresponds to DDL, this is the PlanVersion end time
apsi.locking.duration.min=00.000.00.00.10.000	Attitude timeline, transition to Locking state, min duration, YY.DDD.HH.mm.ss.SSS, avoid +INF
apsi.locking.duration.max=00.000.00.00.10.000	Attitude timeline, transition to Locking state, max duration, YY.DDD.HH.mm.ss.SSS, avoid +INF
apsi.locked.duration.min=00.000.00.00.10.000	Attitude timeline, transition to Locked state, min duration, YY.DDD.HH.mm.ss.SSS, avoid +INF
apsi.locked.duration.max=00.000.00.00.10.000	Attitude timeline, transition to Locked state, max duration, YY.DDD.HH.mm.ss.SSS, avoid +INF
apsi.makepic.duration.min=00.000.00.00.04.000	Camera timeline, transition to makePic state, min duration, YY.DDD.HH.mm.ss.SSS, avoid +INF
eapsi.makepic.duration.max=00.000.00.00.04.000	Camera timeline, transition to makePic state, max duration, YY.DDD.HH.mm.ss.SSS, avoid +INF

8.3 APSI DDL file

The property is listed in application.properties above.

8.4 Orekit library initialization folder

The property is listed in application.properties above. It is a folder for the necessary Orekit files. The Autonomy App needs only UTC-TAI.history, but other files may be copied here if needed.

8.5 Classifier models

The property is listed in application.properties above.

9 Autonomy App output

The java log (see logging.properties in source) is located in the application root. It gives an overview of what's going on with the Autonomy App. Dedicated logging is produced to the planning, images, and monitoring folders, configured in application.properties (see above).

9.1 "planning" folder

Contains the PDL (input to planner) and planner output files. The file names use a pattern:

Request_<timestamp>_{TakeImage,Scout,Survey}_iter<n>_problem.pdl
Request_<timestamp>_<request_identifier>_iter<n>_plan.txt

The timestamp denotes the request submission time. "iter1" is the original request planning, "iter2" the first re-planning, etc. Example:

Request_2020-06-19T12:11:23_TakeImage_iter1_problem.pdl
Request_2020-06-19T12:11:23_TakeImage_iter1_plan.txt

9.2 "images" folder

Image files use a pattern:

Request_<timestamp>_TakeImage_<latitude,longitude>_activityInstanceId<activityInstanceId>_iter<n>.jpg

The timestamp denotes the request submission time.

"iter1" is the original planning, "iter2" the first re-planning, etc. Example:

Request_2020-07-03_09:27:57_TakeImage_10,51_activityInstanceId1_iter1.jpg

9.3 "monitoring" folder

This folder contains four files, which correspond to the Published activity updates, Published planning requests, Published plan versions, and Published plan statuses tabs/displays in CTT. The files use a pattern:

AutonomyAppSession_<timestamp>_PublishedActivityUpdates.txt
AutonomyAppSession_<timestamp>_PublishedPlanStatusUpdates.txt
AutonomyAppSession_<timestamp>_PublishedPlanUpdates.txt
AutonomyAppSession_<timestamp>_PublishedRequestUpdates.txt

The timestamp denotes Autonomy App start time.

The log files use the same columns and format as in the Activity/Request/Plan/PlanStatus update display tabs in CTT.

10 Autonomy App deployment options - overview

There are multiple setups to run the Autonomy App.

10.11) SDK / Simulator / App started from Supervisor

This is the baseline option for running the Autonomy demo in development environment. The steps are described in detail below.

10.22) SDK / Simulator / App started from CLI

This option is similar to the first, except that the App is started from CLI. The main difference is that the App log goes to shell window instead of Supervisor tab in CTT. The steps are the same is in the first option, except the Autonomy App part (step 2).

$ cd <project_root>/sdk/sdk-package/target/nmf-sdk-2.0.0-SNAPSHOT/home/mp-autonomy-demo

Add one more property (in addition to the other ones) to provider.properties. It is the Supervisor URI, which is logged to the shell window where the Supervisor was started:

esa.mo.nmf.centralDirectoryURI=maltcp://10.0.2.15:1024/nanosat-mo-supervisor-Directory

Start the Autonomy App, and proceed with Simulator and CTT:

$ ./start_mp-autonomy-demo.sh

10.33) SDK / no Simulator

This option is impractical for the Autonomy demo. The planned activities in realistic scenario are spaced hours or days apart. In order to fast-forward time to next planned activity it would be necessary to change the System time. The same effect is achieved by Simulator by changing the time provided by the Platform Clock service.

10.44) OPS-SAT-like environment

The options so far used the development environment in "nanosat-mo-framework". This option needs in addition the "nmf-mission-ops-sat" repo. Follow the steps:

https://nanosat-mo-framework.readthedocs.io/en/latest/opssat/testing.html

The CTT connects to Ground MO Proxy (see the first diagram). The steps are summarized in Appendix.

Note! Requests can be submitted only with direct contact with satellite.

Note! Published events can be refreshed from COM Archive.

11 Demo steps: SDK / Simulator / App started from Supervisor

The first option to run the Autonomy App is in the NMF development environment, using Supervisor and platform services simulator. This is intended for testing planning, classification, inspecting logs. Processes are started in separate shell windows.

11.11) Start supervisor sim

$ cd <project_root>/sdk/sdk-package/target/nmf-sdk-2.0.0-SNAPSHOT/home/nmf/nanosat-mo-supervisor-sim
$ nanosat-mo-supervisor-sim.sh

11.22) Configure autonomy app

This section lists the minimum to get the App started. See the "Configuration" section for all options.

$ cd <project_root>/sdk/sdk-package/target/nmf-sdk-2.0.0-SNAPSHOT/home/mp-autonomy-demo

In provider.properties:

esa.mo.com.impl.provider.ArchiveManager.droptable=true
esa.mo.nmf.app.systemTimeProvidedByPlatformClockService=true

11.33) Start spacecraft simulator GUI

The simulator provides random images an allows fast-forwarding simulated system time. The latter is convenient, since the TakeImage activities can be days apart.

$ cd <project_root>/sdk/sdk-package/target/nmf-sdk-2.0.0-SNAPSHOT/home/nmf/spacecraft-simulator-gui
$ spacecraft-simulator-gui.sh

Go to "Camera Simulator Settings" tab. Set the "cameraSim.imageMode" property to "Random". Select the "cameraSim.imageDirectory" property. Browse to <project_root>sdk/examples/space/mp-autonomy-demo/example-images/.

During the CTT demo the system time can be changed using the "Edit Header" button. Set the "End Date" to future. "Start Date" will be the new system time beginning.

11.44) Start CTT GUI

$ cd <project_root>/sdk/sdk-package/target/nmf-sdk-2.0.0-SNAPSHOT/home/nmf/consumer-test-tool
$ cd consumer-test-tool.sh

11.55) Start Autonomy App in CTT

Paste the following address to "Directory Service URI" and press "Fetch Information" (this is the Supervisor URI from step 1, visible in shell window output):

maltcp://10.0.2.15:1024/nanosat-mo-supervisor-Directory

Select "nanosat-mo-supervisor" in left pane and press "Connect to Selected Provider". A new "nanosat-mo-supervisor" tab is created.

Go to "nanosat-mo-supervisor" tab. Select "mp-autonomy-demo" and press "runApp". The log window shows "INFO: MPCameraController initialized!" and "INFO: MPScoutController initialized!".

Go back to first tab ("Communication Settings (Directory))" and press "Fetch Information" to refresh.

Select "App: mp-autonomy-demo" in left pane and press "Connect to Selected Provider". A new "App: mp-autonomy-demo" tab is created.

11.66) Start Scenario 1 and 2 in CTT

Scenario 1 is started by submitting a "TakeImage" request.

Scenario 2 is started by submitting a "ScoutRequest" request.

See the CTT section.

11.77) Monitor the Autonomy App in CTT

The CTT can be used as a mini flight control center to monitor the Autonomy App execution. See the CTT section.

12 CTT: MP services / Autonomy App

This section gives an overview of MP support in CTT. Start the CTT and and the Autonomy App as described above.

Navigate to the "App: mp-autonomy-demo" tab.

This section explains the tabs in the CTT Autonomy App. Each tab corresponds to MO service or COM object monitoring (the "Published" tabs). The following is a brief description of the tabs.

12.1 Submit a request / Planning Request Service

Go to the "Planning Request service" tab. In the convenience drop-down box picks a request type. Select "TakeImage" (scenario 1) or "ScoutRequest" (scenario 2). Click "submitRequest".

Identifier box opens, give it a description name and "Submit".

"RequestVersionDetails" box opens. Specify parameters or submit with default values:

TakeImage request (scenario 1)

Position (lat,long) is specified in ActivityNode stucture (i.e. ImageRequest requests several TakeImage activities, position is part of activity). Follow the steps: "activities" | "activities" | "Edit" | "SimpleActivityDetails" | "argSpecs" | "Edit" | "argSpec" | "positionExpression". In the "value" field enter the latitude and longitude separated by comma, e.g. 10.001,51.002. Click "Submit" on every dialog box to get back to "RequestVersionDetails" dialog. **Note!** Update the description to include the coordinates.

Request validity is specified in "validityTime" | "TimeWindow". The start and end times have to be entered as longs. The default start time is now, and end time one month forward.

ScoutRequest request (scenario 2)

Position (lat,long) is the scouting center coordinate. Multiple positions are specified withing ScoutRequest. Follow the steps: "arguments" | "Edit" | 0 argument | "Edit" | "argValue" | "Edit". In the "value" field enter the latitude and longitude separated by comma, e.g. 10.001,51.002. Click "Submit" on every dialog box to get back to "RequestVersionDetails" dialog. **Note!** Update the description to include the coordinates.

Interest classification. "arguments" | "Edit" | 1 argument | "Edit" | "argValue". Valid values are "Volcano", "Reef", "Human-made", "Land", "Water". They are all set as defaults. Delete the ones not needed.

Request validity is specified in "validityTime" | "TimeWindow". The start and end times have to be entered as longs. The default start time is now, and end time one month forward.

After the request is submitted, monitor progress using the Published activity updates, Published planning requests, Published plan versions, and Published plan statuses tabs.

The Published activity updates tab gives the time of the next planned activity execution. If simulator is used, it makes sense to fast-forward the system time.

12.2 Published planning requests

This tab displays planning request status updates in chronological order. The same information is duplicated in a log file in the on-board "monitoring" folder (see the "Configuration" section). There is a Refresh button to load/refresh events after the Ground MO Proxy syncs to on-board Archive after contact with satellite is established.

The MP request statuses have the following semantics, listed in transition order:

REQUESTED - the TakeImage or ScoutRequest is received by the Planning Request service
ACCEPTED - the request (and its activities) passe validation
REJECTED - error occurred, e.g. invalid coordinates or validity period
PLANNED - the activities contained in request have been planned
SCHEDULED - the activities contained in request are submitted for execution
SUSPENDED - this state is not used
EXECUTING - at least one requested activity is in EXECUTING state
COMPLETED - all requested activities are in COMPLETED state
CANCELLED - this state is not used
STOPPED - this state is not used
FAILED - runtime exception has occurred

12.3 Published activity updates

This tab displays activity updates in chronological order. The same information is duplicated in a log file in the on-board "monitoring" folder (see the "Configuration" section). There is a Refresh button (see "Published planning requests" above).

The MP activity statuses have the following semantics, listed in transition order:

UNPLANNED - the activity has been instantiated from request. Note! MP request contains parameters, from which activities are instantiated. If classifier decides that activity has to be re-planned, then the activity state is set to UNPLANNED
PLANNED - the activity has been planned
DROPPED - this state is not used
COMMITTED - the activity has been released to execution
SUSPENDED - this state is not used
EXECUTING - the activity is executing
COMPLETED - the activity has finished execution
STOPPED - this state is not used
FAILED - runtime exception has occurred

The "information" field shows extra information, such as classifier output "Cloudy/Clear/Interesting/Not interesting".

Double clicking opens an Activity COM object, in this case ActivityUpdateDetails. Navigating the "Retrieve Related" opens the ActivityInstance and from there "Retrieve Source" opens the ImageRequest.

This tab also shows the next PLANNED activity waiting for execution. The simulator time can be fast forwarded in Simulator, the 'Edit Header' button on top (**note!** 'end date' must be set to future as well). The time factor acceleration is not recommended. x1000 speeds through 24h in 86s, which misses visibility windows.

12.4 Published plan versions

MP Plans have plan versions.

This tab displays plan version updates in chronological order. The same information is duplicated in a log file in the on-board "monitoring" folder (see the "Configuration" section). There is a Refresh button (see "Published planning requests" above).

MP PlanVersion corresponds to APSI plan.

12.5 Published plan statuses

This tab displays plan status updates in chronological order. The same information is duplicated in a log file in the on-board "monitoring" folder (see the "Configuration" section). There is a Refresh button (see "Published planning requests" above).

The MP plan statuses have the following semantics, listed in transition order:

DRAFT - the MP plan has been created
RELEASED - planner has completed
SUBMITTED - the MP plan has been submitted to execution
ACTIVATED - the plan execution has started
TERMINATED - all activities contained in plan have reached COMPLETED or FAILED state

12.6 Plan Information Management Service

View and inspect MP service configuration by pressing "listRequestDefs" and "listActivityDefs" buttons.

12.7 Action Service

Press the "listDefinition(*)" to view and inspect the TakeImage and SetAttitude actions.

12.8 Archive Manager

Press "getAll" button any time to see the Archive contents.

12.9 Using simulator time

See 'Published activity updates' above.

13 Design Overview: Scenario 1

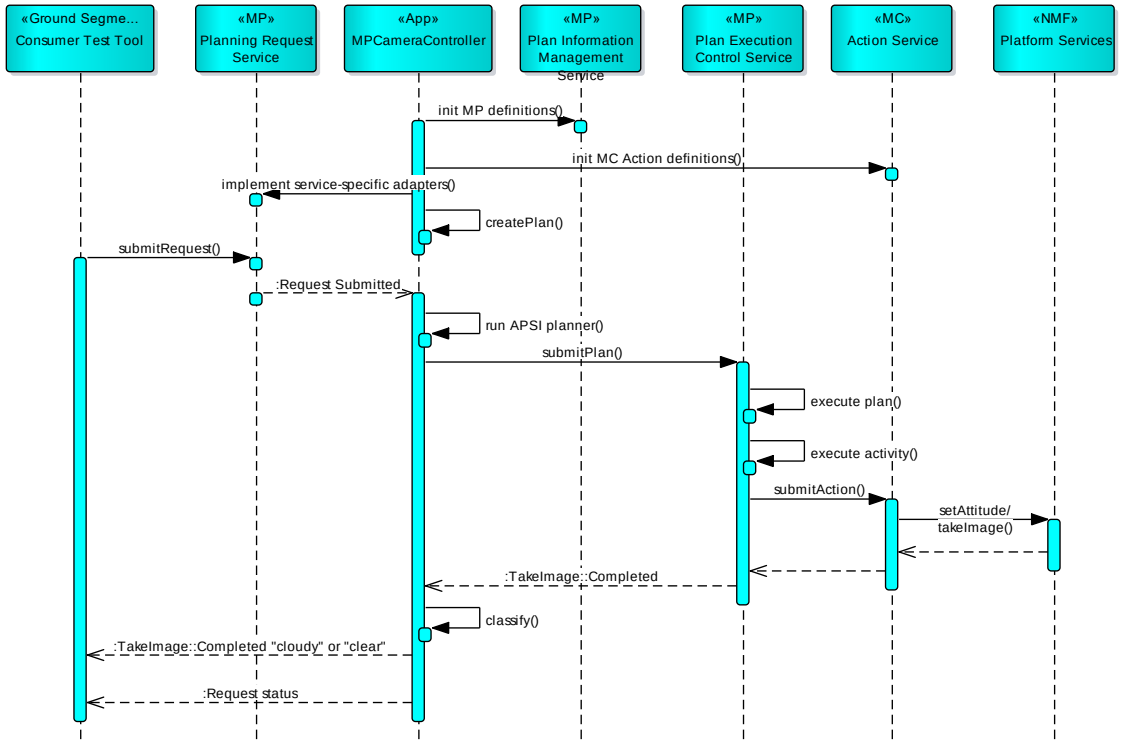
The goal of the Autonomy experiment Scenario 1 (provided by the APSI team) is to take clear images using on-board camera. The initial request is submitted from ground. To make the experiment more interesting for MP services, allow multiple coordinates per request. The image requests are received on-board, planned, scheduled, and executed. On-board classification algorithm evaluates the images for clear/cloudy status. If cloudy then re-plans the request. The image request returns with a Completed status once all images are classified Clear or a time limit is reached. Time limit is specified within the original request.

The class with main method is MPAutonomyDemo. It hosts both scenarios.

Autonomy App overview is described in Diagram. The system is configured using Plan Information Management service. The main flow and feedback chain uses the Planning Request, Plan Execution Control, Action Service and Platform services. Omitted from the diagram are Planning Process Management (may be implemented using MC services), Plan Distribution Service (used for plan status distribution), Plan Edit Service (not needed here), Archive Service (to reduce clutter).

There is one plan. Planning is triggered when new request arrives or an image acquisition needs to be re-planned (the image was cloudy). During planning the existing planned items are considered as flexible facts, meaning their execution window can be slightly adjusted by the planner. The Plan Execution units are SetAttitude and TakeImage Activities. Planned Activity execution triggers a corresponding MC Action. MC Actions call operations from Platform services, such as Camera service TakeImage action. The system is coordinated by MPCameraController, which also initiates re-planning of activities.

The TakeImage activity resulting in a cloudy image is re-planned preserving the activity instance ID. This was a design decision to facilitate planning-replanning cycle demonstration (REQ-2) using the CTT.



Note. The use of MC Action service is not needed for Activity execution in NMF. Activities may call Platform services directly. The MC Action service is used as an exercise in executing MP Activities, for cases where the controlled system is remote.

Note. The COM Archive is used as an event hub. It keeps the run-time model (e.g. activity update objects), and notifies interested components of status updates, for example.

Note. Times are in GMT.

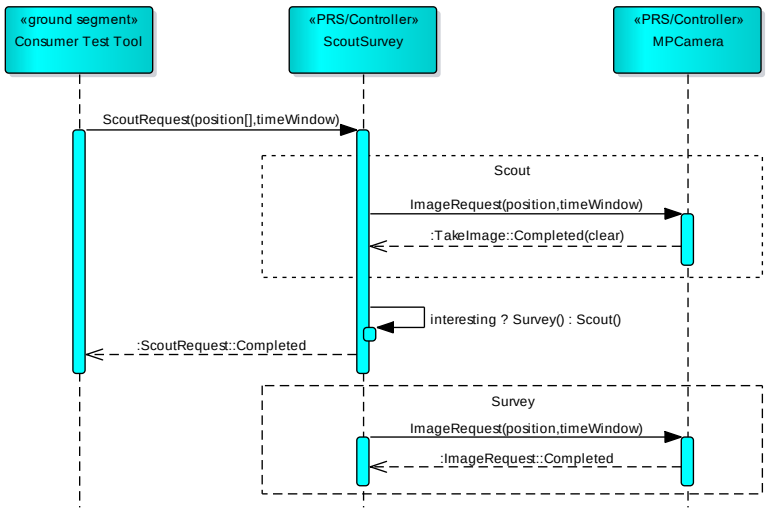
14 Design Overview: Scenario 2

The autonomy Scenario 2 is a scout/survey setup, which uses Scenario 1. ScoutRequest is submitted from CTT. The ScoutRequest has no embedded activities (the TakeImage activities are generated on-board). Instead, the ScoutRequest contains the center coordinates of areas to be scouted. In a simplified implementation there is one image to be taken in any scouted area. The second argument of ScoutRequest is a time window for scouting and survey. ScoutRequests are processed onboard by ScoutSurveyController. Camera shooting is delegated to MPCameraController, which is the system behind Scenario1. For each Survey coordinate the ScoutSurveyController submits an ImageRequest (containing one TakeImage activity), processed by MPCameraController. MPCameraController itself takes pictures until it achieves a clear picture. The resulting TakeImage activity status is monitored by ScoutSurveyController. The MPCamera's ImageRequest status is completed when the image is clear or there was no clear image within the time window. In case the image was clear, the ScoutSurveyController runs a classifier to search for volcano, reef, etc. Depending on outcome, the Scout phase is repeated (time window permitting), or additional picture is taken, now in the Survey phase.

The ScoutRequest is completed, when timeWindow elapses.

The Scenario is demonstrated using CTT, where the Activity monitoring shows "scout" and "survey" extra info for TakeImage activities.

Note. CTT shows the original scout request and ImageRequest sent by SurveyController.



15 Detailed Design

This section and the component descriptions refer to Scenario 1.

15.1 Autonomy App design choices

The business logic of the Autonomy experiment (MPCameraController on Diagram) lies within re-planning, or resubmitting of cloudy images.

About "resubmitting" a request. A goal was to demonstrate the re-planning cycle, e.g. show a status trail of TakeImage action something like "Planned-Executed-Cloudy-Replanned-Executed-Clear" (REQ-2). There were two options: re-submitting using a request, and recycling a TakeImage activity (the latter is implemented).

- Submitting a re-planning request. A consideration was to distinguish the original image request from re-planning image request. For example a client submits a request, and is interested in it's status, but not re-planning (i.e. application internals). This can be achieved with an attribute in ActivityUpdate (for status filtering downstream, essentially bypassing model), or a dedicated request type, "ImageReplanningRequest". But, ImageReplanningRequest instatiates a new TakeImage activity with a new ActivityInstanceID. This creates overhead of matching the replanned TakeActivity instanceID with the original TakeImage. Another consideration was ClearImageRequest/TakeClearImage request/activity pair, matched with ImageRequest/TakeImage (essentially two chained request nodes). The latter used in re-planning cycle. Again, correlating the original and re-planning requests/activities creates overhead. However, the overhead is more manageable, if there is a self-imposed constraint of allowing one TakeImage activity per request.
- Sending an activity instance from Terminated ("cloudy") state to Planned. This approach has the advantage of preserving the actionInstanceID, and therefore activity monitoring captures the re-planning cycle without any further overhead.

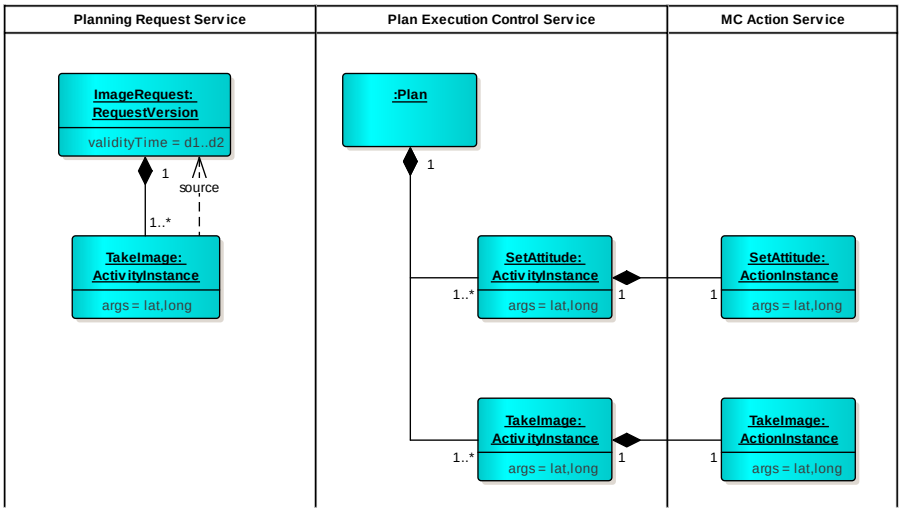
The busines logic is implemented within MPCameraController. Alternative locations for the business logic were considered in Activity execution and Action execution:

- Business logic in Activity execution. As discussed above, a goal was to delegate MP activity execution to MC Action service.
- Business logic in Action execution. There are two approaches: a procedure, and action sequence.
    - Procedure. Takes the picture, classifies, resubmits if cloudy. This option was discarded as too monolithic.
    - Action sequence of TakeImage, Classify, ResubmitIfCloudy. The complexity is passing the output of TakeImage to Classify, and from Classify to Resubmit. This is clumsy using the MC service. The action is submitted using Action service, delivery and execution progress is monitored using ActivityTracking service (no return values). Return values can be implemented using Parameter service. Alternatively, Action service may update some COM object in Archive (e.g. to send a return value, or a status), which then notifies interested parties. Dismissed as too complicated.

About moving controller to ground segment. A design consideration was to show that the business logic can be located remotely, such as ground segment. This is possible, since MPCameraController interacts with Autonomy App components using MO interfaces. However, this may not be practical since there is communication with on-board COM Archive.

15.2 Service Objects

Figure shows the main MP instance object types, attributes and relationships.



The main service objects are ImageRequest and TakeImage activity. The attributes are validityTime and image coordinates. ImageRequest contains one or more TakeImage activities.

SetAttitude is an activity that is inserted by planner. This is required for satellite orientation prior to image capture.

Activity instance IDs are created on-board after receiving ImageRequest.

TakeImage and SetAttitude activities are executed using SetAttitude and TakeImage MC Actions. The Action implementations call setAttitude and takeImage Platform Service operations. The Actions have stage count set to three, which helps in debugging. The worst case locking time for SetAttitude is considered 20s, which is accounted for by the planner.

The source links refer to the creator of the object. TakeImage activity instance has a "source" link to ImageRequest request.

MP ActivityUpdate objects. The startTrigger and endTrigger fields in some states (PLANNED) correspond to APSI start interval, because MP model does not allow for start interval. In other states (EXECUTED) the endTrigger represents the execution (start) time. The 'errInfo' field is used to convey meaningful information to the CTT operator, such as 'TakeImage(10,51)', this is presented in the 'Information' field in CTT 'Published activity updates'. A longer alternative was to use Arguments field.

15.3 COM Archive

COM Archive is the NMF implementation of COM Archive Service. The MP services interact with COM Archive using MPArchiveManager (added in predecessor work), which is a front-end, typed by Request, Activity and other MP top-level items.

Every MP service component in the Autonomy App interfaces with the COM Archive. This relationship is omitted from diagrams for brevity purposes. The NMF implementation of MP services archives all COM objects.

Note. The COM Archive keeps an uptodate state of the model, and sends update notifications. The status updates are created using a pattern. For example, MPCameraController creates an ActivityUpdateDetails object with status Planned for TakeImage activity, and adds the ActivityUpdate COM object to Archive. The Archive stores the object and publishes an ObjectStored event. Plan Execution Control listens for COM objects (using COM Event Service) with TakeImage identity, and publishes activity updates, which were registered using the monitorActivities operation in Plan Execution Control Service.

15.4 Planning Request Service

The Planning Request Service is the main entry point to the application. It receives the ImageRequest, archives it and publishes the request Requested status. This is the default NMF implementation of submitRequest operation. The application flow continues in MPCameraController.

15.4.1 Provided interfaces

Planning Request Service

15.4.2 Consumed interfaces

COM Archive Service. The PRS submitRequest operation stores the submitted RequestVersion (this term may change to "RequestInstance" in future MP model) objects to Archive. New RequestStatusUpdate objects are stored to Archive, which publishes ObjectStored event.

15.4.3 Initialization

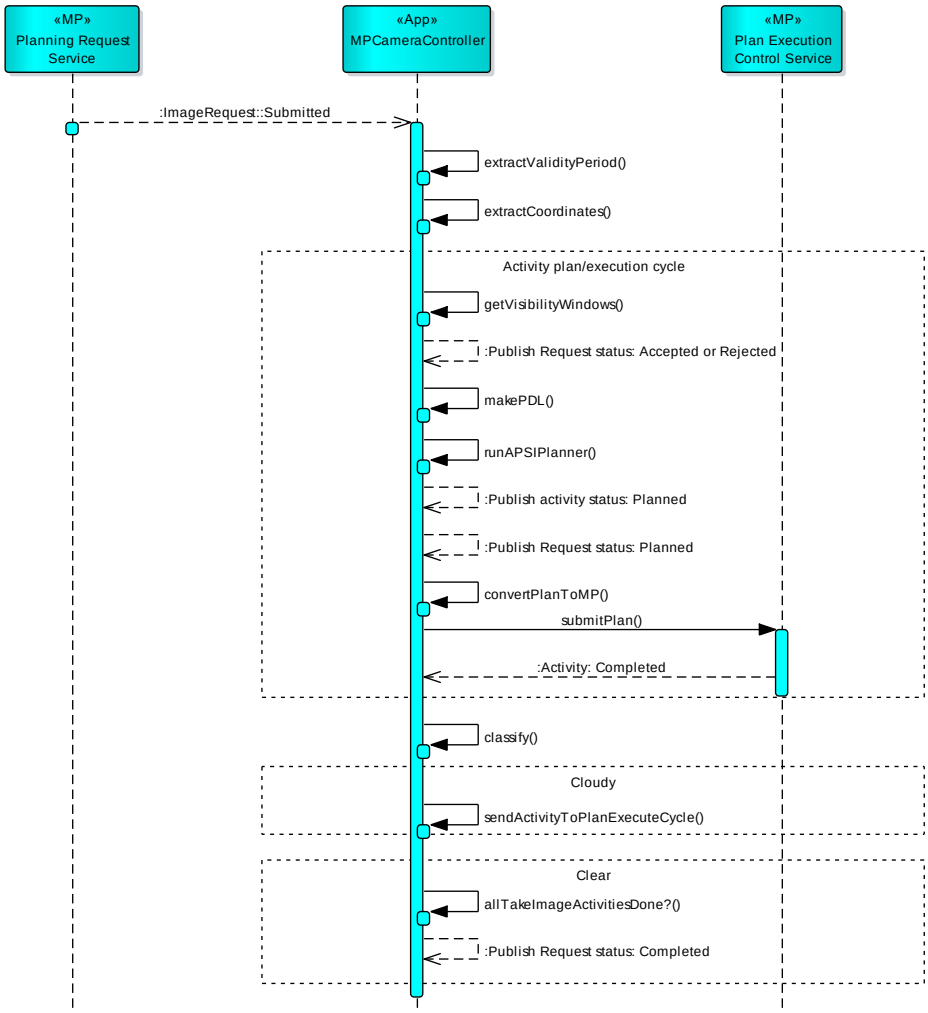
None

15.4.4 Other components

None

15.5 MPCameraController

MPCameraController coordinates the plan/execution chain of image requests. In a nutshell it extracts request validity time period and image target coordinates from ImageRequest, calls the flight dynamics routine that finds coordinate visibility windows, interfaces with the APSI planner, and submits the resulting MP plan to Plan Execution Control service. The steps are summarized in the diagram. The diagram initialization steps, which are described below in components specification.



Note. MPCameraController was initially in the submitRequest method of Planning Request Service. Managing the two request types ImageRequest and ClearImageRequest separately (MPController and ClearImageController) keeps the source code better organized. It also leaves the possibility of deploying ClearImageController to ground segment for demonstration purposes.

15.5.1 Provided interfaces

None

15.5.2 Consumed interfaces

Planning Request Service. Monitors ImageRequest Submitted status updates. Initiates plan/execution chain on ImageRequest Submitted status.

Plan Information Management Service. Configures ImageRequest request template identity, template, identity and version; TakeImage and SetAttitude activity identity and definition.

Action Service. Configures TakeImage and SetAttitude action definitions.

Plan Execution Control Service. Submits MP plan for execution.

COM Archive Service. New RequestStatusUpdate and ActivityUpdate COM objects created by MPCameraController are stored to Archive, which publishes ObjectStored event.

15.5.3 APSI components

APSI components are contained withing the APSI jar. From the APSI jar the Autonomy App uses the planner, clear/cloudy classification, interest classification, visibility window calculator.

Note. The classifiers and visibility window are not APSI components per se, they are distributed by APSI jar.

Note. The classifier code is not in the APSI jar. It is located in <project_root>/sdk/examples/space/autonomy/src/main/java/esa/mo/apps/autonomy/classify

The APSI planner is used as a library. The planner has two inputs. The DDL (Domain Description Language) input file is static. It describes the planner universe. In the Autonomy experiment the planner models MissionTimeline, GPSSystem, Attitude, Camera, and Classification. Each timeline has a set of states that it transitions through. The planner finds several solutions that transition between the states. The Autonomy experiment picks the first solution. The second input is the PDL (Problem Description Language) file. The PDL states the problem description (the target coordinates and visibility windows). In a re-planning cycle the PDL declares the not yet executed activities as (flexible) facts.

Maximum number of planning solutions is hard-coded to be one, since we take the first always (MPCameraController, PLANNER_MAX_NUMBER_OF_SOLUTION).

DDL. The MP part of the Autonomy app uses transitions to the Attitude.Locking and Camera.makePic states, which it translates to SetAttitude and TakeImage MP activities. The other timelines are not considered. This means that re-planning declares as facts only the above Attitude and Camera transitions.

Worst case locking and setAttitude. The AutonomousADCS.setDesiredAttitude takes a "duration" parameter, which is controlled from application.properties (adcs.setattitude.duration, read the comment there). The duration parameter is taken as sum of time to set and maintain the attitude.

Coordinates. The APSI planner uses integer coordinates. In order to simulate precision to three decimal places use a x1000 multiplier. E.g. a coordinate10.001,51.002 is transformed to 10001,51002 for the planner. Note that this translation is already applied in the DDL file.

Visibility Window calculator. Takes GPS.getTLE() as input. See MPCameraController.getVisibilityWindows. It delegates computation to APSI jar. Visibility windows are input to PDL.

APSI plan. The plan lists the time interval for activity/transition and min/max duration of the activity/transition. Autonomy App uses the APSI plan structures vs plan.txt file to translate to MP structures.

MPCameraController.imageRequested: this method processes the submitted request. The requested coordinates ('goals' in code) and already planned activities ('plannedTimeline') are merged into single PDL. The 'solveGoals' method delegates to MPConverter.convert, where the PDL is generated, by writing the requested goals and facts to a physical PDL file. There is a similar to 'toPDL' method in APSI framework, that converts plan to PDL. This knowledge came too late in development, besides, the PDL generation was still needed for submitted targets. The APSI plan is merged to executing plan in 'mergePlannedActivities' call. If an activity is already on plan (compares activity instanceIDs), then it is not inserted (it was presented to planner as a fact, not a as a new goal).

About MP activity instance IDs. The TakeImage activity instanceID is generated by 'TakeImageActivity.create' call (MPCameraController.imageRequested). The instanceID is an argument in Camera timeline. It is propagated by planner from PDL to plan. This is for the MP structures to track the activity lifecycle. The Attitude timeline on the other hand does not take instanceID, since Attitude timeline is generated from Camera. The reasoning is that there can be one Attitude change followed by several Camera actions. This means there is no way to match SetAttitude activity instanceID before and after the planner (there is no way of knowing, if SetAttitude was presented as fact to planner, or not, in which case it was first planning). An implemented workaround in merging SetAttitude activities to execution engine ('mergePlannedActivities') is to match SetAttitude activities by start/end time (since there is no

activity instanceIDs). Also, account for the possibility of re-planned SetAttitude activities to be within the range of original SetAttitude, in case the planner narrows a flexible Attitude fact.

Configuration. See the "Configuration" section.

PDL and plan files folder. See the "Autonomy App output" section.

Image files folder. See "Autonomy App output" section.

Image classifiers are called in MPCameraController.classifyImage (cloudy/clear) and MPScoutController.classifyImage (interest). The interest classification gives a probability of volcano, reef, human-made, land, and water. If any of those are flagged then the Survey phase is entered.

15.5.4 Initialization

Creates identities and definitions in using Plan Information Management and Action Service.

15.6 Plan Information Management Service

The Plan Information Management Service is used for adding request and activity identities and definitions for ImageRequest request, and TakeImage and SetAttitude activities.

15.6.1 Provided interfaces

Plan Information Management Service. This is an out-of-box NMF implementation, which adds the definitions to COM Archive, where they are accessible to other MP services.

15.6.2 Consumed interfaces

None

Initialization

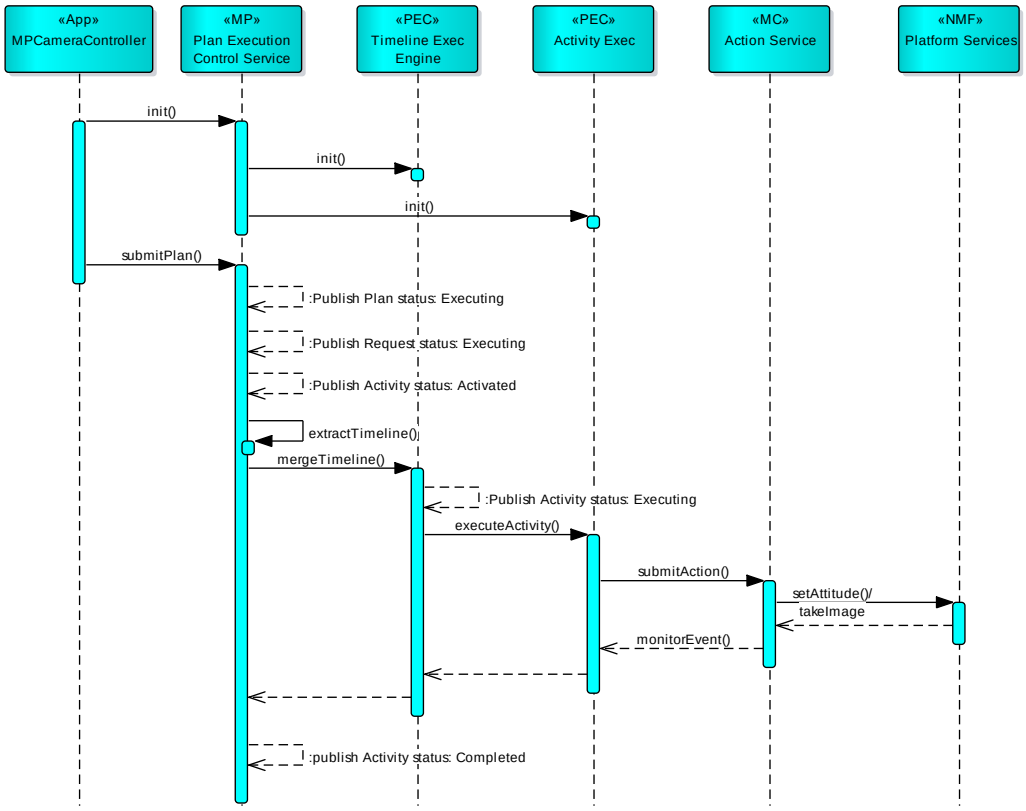
None

15.6.3 Other components

None

15.7 Plan Execution Control Service

The Plan Execution Control Service is currently being specified by CCSDS-MP working group. In scope of this work a minimal PEC service is implemented in NMF, that allows submitting a plan for execution and monitoring plan and activity statuses. As any other NMF MP service implementation, the PEC service has default implementation and adapters for service operation over-rides. Execution control functions such as start, stop, pause, resume are not implemented (pending CCSDS-MP WG). Instead a single (master) plan is constantly being executed, this is Autonomy App specific behaviour. Plan increments are merged using the submitPlan operation adapter (this is specific behaviour). Plan execution takes advantage of the fact, that the activities are time-tagged. A simple plugin execution engine polls every second (configurable to any interval) for planned activities to execute.



15.7.1 Provided interfaces

Plan Execution Control Service. This is the fifth MP service being added to NMF Mission Planning implementations. The service specification is work in progress at the time of this implementation. The provided functionality is submitPlan and plan/activity execution monitoring.

15.7.2 Consumed interfaces

MC Action Service. The planned activities call submitAction with SetAttitude and TakeImage action instance details.

COM Archive Service. The RequestStatusUpdate and ActivityUpdate COM objects created by PEC are stored to Archive, which publishes ObjectStored event.

15.7.3 Initialization

None

15.7.4 Other components

Timeline Execution. A stack of planned items by execution time. Execution time is polled at an interval of one second by default. This component is MP agnostic. It has three classes:

- esa.mo.mp.impl.exec.TimelineExecutionEngine. Methods submitTimeLine, start, stop, setTickInterval. Uses java.util.concurrent.ScheduledExecutorService.
- esa.mo.mp.impl.exec.TimelineItem. Fields earliestStartTime, latestStartTime, String itemId, ItemCallback callback (code that runs goes here).
- esa.mo.mp.impl.exec.ItemCallback: Methods execute (code to be executed), missed (handler for executions that are skipped, e.g. tick interval too large).

Activity Execution. This is MP Activity execution code. Item that is popped from TimelineExecution is converted to MC Action instance and calls submitAction in MC Action

service. There is Activity to Action lookup (the "register" method). Relevant classes are ActivityExecutionEngine and ExecutableActivity.

15.8 MC Services

The MC Services are implemented by NMF and are available to all Apps. The Autonomy App consumes the MC Action service.

15.9 Platform Services

The Platform Services are implemented by NMF and are available to all Apps. The Autonomy App consumes the following Platform services:

ADCS Service: called by SetAttitude action.

Camera Service: called by TakeImage action.

16 Source code and "mission-planning" branch

The source code is located in "mission-planning" branch of "nanosat-mo-framework" git project. The "mission-planning" branch contains MP services implementations done in the predecessor project. This work added Autonomy experiment Scenarios 1 and 2. For eventual merge to "dev" branch it is recommended to first rebase "mission-planning" onto "dev" and run the tests and demo. Rebasing onto latest "dev" has been done like this:

```
$ git checkout mission-planning
$ git rebase -Xtheirs --no-ff --onto dev <old-dev-hash>
```

The source code is divided into framework code ("core", general MP services provision) and application code ("sdk", MP services demo App, Autonomy Scenario 1 and 2 App).

16.1 core

Located in "core/mo-services-impl/ccsds-mp-impl/" (esa.mo.mp.impl.*). This is the general MP services implementation in NMF. In scope of this work there was PlanExecutionControlProviderServiceImpl added to MPServicesProvider. A simple time-tagged activity execution is implemented in sub packages:

esa.mo.mp.impl.exec: timeline execution, time-tagged planned MP activities are polled for execution time at regular interval. **Note** run-time throws InterruptedException in java.util.concurrent.locks.AbstractQueuedSynchronizer. This is expected.

esa.mo.mp.impl.exec.activity: planned MP activity execution that delegates to MC Actions.

Platform.Clock service. The motivation for such service came from MP Plan Execution Control Service implementation, where activities are polled for execution every second. In order for this to work with Simulator, where time can be shifted and accelerated, the system time needs to be provided by the simulator. Thus a Clock service is defined in ServiceDefPLATFORM.xml with methods "getTime" and "getTimeFactor". The Clock is implemented in esa.mo.helpertools.clock.SystemClock. If USE_PLATFORM_CLOCK_PROPERTY is set (esa.mo.nmf.app.systemTimeProvidedByPlatformClockService=true in App provider.properties file) then system time is provided by Platform Clock service. Else return System.currentTimeMillis(). NB! CTT is not migrated to Platform Clock (about four places to update). NB! The simulator provides Clock service. Since the simulated time is updated every second, a max error with x1000 time factor is about 16 minutes (1000s).

TransactionsProcessor::insert - fix to allow retry on failed insert to COM Archive insert queue.

16.2 sdk

The supporting Autonomy code is in "sdk/examples/space/autonomy/". This package contains the APSI jar and classes that interface with it.

The Autonomy App is in "sdk/examples/space/mp-autonomy-demo/". The entry point to Autonomy App is MPAutonomyDemo, which delegates to MPAutonomyDemoAdapter (App specific MP service behaviour). Sub packages in mp-autonomy-demo:

esa.mo.nmf.apps.mp.activity: SetAttitudeActivity, TakeImageActivity implementations.

esa.mo.nmf.apps.mp.controller: MPCameraController - App coordination and Autonomy business logic.

17 Documentation

ReadTheDocs compatible documentation is located in <project_root>/docs/source/mpservices.
This TN is linked as downloadable PDF in autonomy.rst.

18 Summary/discussion

The Autonomy App findings are discussed below, grouped by project goals.

18.1 REQ-1 MP services and data model demonstration

The NMF Autonomy App demonstrates closed loop plan-execute cycle. Each main component provides one MP service. The main components interact with each other using MO service interfaces, meaning the components may theoretically be deployed in different segments. However, the Autonomy App uses on-board COM Archive for run-time data and for Update objects notification, which makes it impractical to mix the components in ground/space.

The MP data model is compatible with COM Archive, since the MP data objects are wrapped with COM model. The MP model was persisted to COM Archive, which acted as data hub. COM Archive consumers may register to receive update notifications, such as Request and Activity status updates. In the Autonomy App the Request updates were distributed by Planning Request service, and the Activity updates by Plan Execution Control service, effectively acting as facade to COM Archive status update notifications.

The telemetry (request, activity, plan statuses) is published by the MP services, and received on ground live when there is direct contact with the satellite. Since the run-time application data was persisted to COM Archive, it is possible to synchronize the ground mirror of the COM Archive (in Ground MO Proxy) with the on-board COM Archive, and then refresh the published events from the (Ground MO Proxy) COM Archive. This allows viewing the published events that were transmitted offline.

MP model features used:
Request COM objects: RequestTemplateIdentity, RequestTemplate, RequestIdentity, RequestVersion, RequestStatusUpdate
Activity COM objects: ActivityIdentity, ActivityDefinition, ActivityInstance, ActivityUpdate
Plan COM objects: PlanIdentity, PlanVersion
Request attributes: validityTime
Data types: Position

Managing the ActivityInstance/RequestVersion run-time relation is not directly supported by the MP model. In Autonomy App it is handled using the COM source link.

APSI planner used an interval for activity start, which is not supported by the MP model. The workaround was to use the "start" and "end" attributes in certain activity states (e.g. PLANNED) to describe the interval.

18.2 REQ-2 On-board autonomy demonstration using MP services

The business logic, that classifies the image and resubmits if cloudy, is implemented in a controller that acts as MP services consumer. This approach works, since the business logic is built on top of plannable image capture system. Autonomy is achieved since the controller is deployed on-board as part of the App.

Delegating planned activity executions to MC Action service is intended for controlling a remote system. The submitAction operation in MC Action service is not a "remote procedure call", and creates overhead in an on-board application, since action is submitted for execution using Action service, delivery and execution progress is monitored using ActivityTracking, and values returned using Parameter service.

In re-planning the Activity IDs are preserved. This choice facilitates the planning-replanning cycle demonstration in the Consumer Test Tool, where the Activity states (UNPLANNED, PLANNED, etc) correspond to the same Activity instance ID.

18.3 REQ-3 APSI components demonstration

The APSI planner models entity transition timelines. Re-planning considers already planned but not executed tasks as facts. It has a static universe description in DDL format and problem description in PDL. The Autonomy App used the spacecraft attitude and camera timelines, in addition the sample DDL modelled the classifier, memory and downlink timelines.

For improved marketing the APSI planner may be distributed in a jar. Another jar may be a simple timeline execution engine.

The first image classifier marked image as cloudy/clear. The second image classifier gives a probability of volcano, reef, land, water, and human-made targets.

18.4 REQ-4 MP Service NMF implementations demonstration

The predecessor project to this work implemented MP services and tools, which were now used in the Autonomy App.

The Autonomy App used the Planning Request and Plan Information Management services added in the predecessor work. This was the first realistic scenario for using the implementations. The default implementations archive the services COM objects. Application specific over-rides are implemented in operation adapters. Building the Autonomy App did not require changes to the already implemented MP services.

Plan Execution Control Service is a new addition to the NMF MP suite. Since the service specification is work in progress, only a submitPlan and monitoring of plan/activities were implemented.

The MPArchiveManager facade to COM Archive (added in predecessor work) is typed by Request, Activity and other MP top-level items. It is a convenience API, that belongs to a future SDK of MP tools.

The predecessor work added an API for managing configuration and run-time relationships not covered by model (see REQ-1 discussion in this chapter). This was another convenience in building the Autonomy App. The implementation, however, defines dedicated configuration COM objects in service specification, which is defined on top of the MP service specification.

19 Appendix A: Demo steps: OPS-SAT-like environment and operational

In the operational phase the Autonomy App is monitored and controlled using the CTT with GroundMOProxy. The setup is described on Diagram 1. The difference between the test environment is communication loss when satellite is not in sight. This means the following:

Requests can be sent when satellite is visible.

Published events are received only when satellite is visible. Events published when satellite is not visible can be retrieved from COM Archive during a pass:

1. Sync the GroundMOProxy COM Archive with on-board COM Archive using a workaround: the COM Archive sync in CTT is invoked by starting any App. For that start the "publish-clock" App in CTT Supervisor. **Note**: Publish-clock app is required for MPAutonomyDemo on-board.
2. In MP Published activity updates, Published planning requests, Published plan versions, and Published plan statuses panels click the **Refresh** button to refresh the events from COM Archive.

Ground MO Proxy archive must be cleared when Autonomy App is restarted. Add line to Ground MO Proxy provider.properties and restart the Ground MO Proxy:

esa.mo.com.impl.provider.ArchiveManager.droptable=true

19.1 Deployment

This is a summary of steps from https://nanosat-mo-framework.readthedocs.io/en/latest/opssat/packaging.html

Clone the "nmf-mission-ops-sat" repo and checkout the "dev" branch.

1) /opssat-package/copy.xml:

Update: MAIN_CLASS_NAME esa.mo.nmf.apps.PayloadsTestApp
→ esa.mo.nmf.apps.MPAutonomyDemo

2) /opssat-package/pom.xml:

Update: publish-clock → mp-autonomy-demo (three places)
Update: expId

3) "mvn clean install -Pexp" in project root.

4) "mvn install -Pground" in project root.

5) /opssat-package/target/nmf-opssat-VERSION-SNAPSHOT/

$ tree -d

Copy /experimenter-package/home/expXYZ → /home

6) /opssat-package/target/nmf-opssat-VERSION-SNAPSHOT/home/expXYZ:

In the "nanosast-mo-framework" repo navigate to /sdk/examples/space/autonomy/src/main/external-resources. Copy the following resources:

* /classifier-models folder

* /orekit folder

In the "nanosast-mo-framework" repo navigate to /sdk/examples/space/mp-autonomy-demo/src/main/external-resources. Copy the following resources:

* application.properties

* /apsi folder

7) /opssat-package/target/nmf-opssat-VERSION-SNAPSHOT/home/expXYZ/provider.properties:

helpertools.configurations.provider.app.user=<change to linux user account>

8) /opssat-package/target/nmf-opssat-VERSION/home/nmf/ground-mo-proxy/

$ sudo ./ground-mo-proxy.sh

9) /opssat-package/target/nmf-opssat-VERSION/home/nmf/supervisor-sim/

Copy platformsim.properties from "nanosat-mo-framework" and edit: (**note** this file not used in real flight)

camerasim.imagemode=Random
camerasim.imagedirectory= /absolute/path/to/nanosat-mo-framework/sdk/examples/space/mp-autonomy-demo/example-images/

$ sudo ./nanosat-mo-supervisor-oppsat.sh

If the supervisor starts with a ConcurrentModificationException, then restart it.

10) Start the CTT and connect to Ground MO Proxy URI

11) If Publish-clock App is used to trigger the Archive sync in GroundMOProxy (by starting the Publish-clock App in Supervisor). Go to /opssat-package/target/nmf-opssat-VERSION/home/publish-clock:

* provider.properties:

helpertools.configurations.provider.app.user=<change to linux user account>

$ mkdir lib - and copy there "publish-clock-2.0.0-SNAPSHOT.jar" from "nanosast-mo-framework" repo

19.2 Operation

See the "Demo Steps: SDK" section.

In CTT connect to the Ground MO Proxy (take the URI from the shell where the Ground MO Proxy was started): maltcp://10.0.2.15:1025/ground-mo-proxy-Directory