General-purpose Payload-oriented Software Architecture for Nano-Satellites

Carles Araguz
Elisenda Bou-Balust
Eduard Alarcón
Nano-satellites have become an affordable alternative for companies, research organizations, and universities to access the space market.

- Either as consumers or as providers.
- Low-cost, short development times.
- Proven to be suitable platforms for:
  - technology demonstration (Bowmeester 2010);
  - a variety of EO and remote sensing purposes (Selva 2012);
  - space research (e.g., GeneSat-1);
  - and many other space applications (e.g., low-power communications, maritime activity surveillance).
Nano-satellite programs: current context

→ Many nano-satellite missions are developed under educational programs.
  ▪ Generally less demanding in terms of accuracy and reliability.
  ▪ A clear sign of the on-going democratization of space.
  ▪ Continuous exploration of science return capabilities → complexity growth.

→ Nano-satellites are envisioned to be favorable for the development of new mission architectures (Banhart 2007):
  ▪ Fractionated spacecraft.
  ▪ Satellite constellations (e.g. PlanetLabs Flock).
  ▪ Federated Satellite Systems.
→ Integration of hardware modules/subsystems.
  ▪ Compliant with the CubeSat standard.

→ Developers ultimately need to write their custom software to control the spacecraft at device- and system-level.

→ Less attention has been placed on software-related issues during the consolidation of the CubeSat era.
  ▪ Software is the final architectural element to achieve a desired functionality.
  ▪ Software characteristics are critical for the mission → its correctness affects the functionality of the spacecraft.

→ Designing proper software architectures:
  ▪ Essential to achieve system-wide quality attributes (e.g. reliability, performance…)
  ▪ Should not be understood as the mere fact of writing functionally correct programs.
This presentation addresses the issues related with the design of software architectures for nano-satellites.

Nano-satellite system and software characteristics

What are the critical aspects in flight sw. design?

Guidelines for nano-satellite software design.

Illustrated with an software architecture.

- Extracted from successful missions and the literature.
- Software qualities.
- System requirements.
- Where developers should focus?
- Generic; mission-agnostic.
- Applicable both in educational and industry contexts.
- Our contribution to address critical aspects.
- The ³Cat-1 software architecture.
Nano-satellite system and software characteristics

- Extracted from successful missions and the literature.

What are the critical aspects in flight sw. design?

- Software qualities.
- System requirements.
- Where developers should focus?

Guidelines for nano-satellite software design.

- Generic; mission-agnostic.
- Applicable both in educational and industry contexts.
- Our contribution to address critical aspects.

Illustrated with an software architecture.

- The 3Cat-1 software architecture.
Nano-satellite system characteristics

→ Reusable platforms:
  ▪ The same platform is used for several generations within the same nano-satellite program.
  ▪ Low-cost.
  ▪ Fast development cycles.

→ COTS components:
  ▪ Non-rad-hard technology.

→ No hardware redundancy.

→ Limited power availability.

→ On-board computers with very low computational resources:
  ▪ Single-Board Computer modules (SBC) for embedded applications.
  ▪ ARM CPU, 40~500 MHz.
  ▪ ~256 MB of RAM or less (e.g. GOMSpace’s NanoMind: 4 MB).

→ Constrained communication links.

→ Usually LEO orbits.

→ Ground-operated: time-tagged commands are uplinked and executed sequentially.
Nano-satellite software characteristics

→ Many approaches to improve system reliability:
  ▪ Process isolation and protected memory areas.
  ▪ FDIR methodology: ESA’s OPS-SAT CubeSat, TU Delft’s DelFFi (Bräuer 2015).
  ▪ Software redundancy: triplicate critical data.
  ▪ Robust communications: prevent unreliable delivery of digital data.
  ▪ Hardware and software-watchdogs.
  ▪ …

→ Some approaches towards modularity and updateability:
  ▪ Dynamically-linked libraries: encapsulate re-usable components.

→ Architectural diversity:
  ▪ Centralized: e.g. CalPoly’s 2nd Gen. Bus (Manyak, 2011).
  ▪ Decentralized/distributed: e.g. AAUSAT3 software (Bønding 2008).
Presentation outline

Nano-satellite system and software characteristics

What are the critical aspects in flight sw. design?

Guidelines for nano-satellite software design.

Illustrated with an software architecture.

- Extracted from successful missions and the literature.
- Software qualities.
- System requirements.
- Where developers should focus?
- Generic; mission-agnostic.
- Applicable both in educational and industry contexts.
- Our contribution to address critical aspects.
- The 3Cat-1 software architecture.

Oct. 27th, 2015
Requirements for next-generation nano-satellite software

→ Where should software engineers focus?
  1. Robustness → software quality.
  2. Software modularity and scalability → software quality.
  3. Autonomy → functionality.

→ Fundamental for nowadays’ nano-satellite software.

→ **ROBUSTNESS:**

**Large spacecraft:**
- ✓ Rad-hard components to help mitigate SEU and SEL.
- ✓ Sophisticated real-time kernels, hypervisors and middleware with flight heritage (e.g, cFS/cFE.)

**Nano-satellites:**
- • SEU and SEL are critical (use of COTS and tightly constrained power budgets).
- • Some can be adopted, although they are usually unknown (especially in university programs). Engineers tend to use other alternatives (e.g, FreeRTOS, std. Linux…)
  - • *Designing* robustness is critical (at an architectural level too.)
Requirements for next-generation nano-satellite software

→ The capabilities of satellite constellations consisting of many (identical or heterogeneous) nano-satellites are promising.

→ Unceasing technology miniaturization is allowing new, smaller, less power demanding and more capable devices and modules potentially increasing the payload capacity of nano-satellites.

→ Due to their low cost, nano-satellite platforms are usually reused. New generations of nano-satellites have the same basic subsystems but host different payloads.

**Modularity and scalability:**

- **Flexible** architectures: easy to update/port when the underlying hardware changes.

- **Scalable and modular** in terms of payload functionality (i.e. payload management capabilities)

- Easy to maintain and fix.
Requirements for next-generation nano-satelitte software

→ **AUTONOMY:**

→ Limited observability:
  - Constrained communication links:
    - Usually in the UHF band.
    - Limited telemetry data rates/volume.
    - Downloading fine-grained state history is unfeasible.
  - LEO orbits: ~5-8 passes per day; ~10 minutes each.
  - Need to be able to autonomously recover from errors and re-plan its activities.

→ On-board data analysis improves science return (e.g. NASA’s EO-1: CASPER)
  - Higher computational capabilities are required.
  - In nano-satellites, e.g. IPEX (CASPER).

→ New mission architectures demand unit-autonomy:
  - Nodes need to proactively cooperate (communicate, exchange resources) to achieve global common goals.
Presentation outline

Nano-satellite system and software characteristics

- Extracted from successful missions and the literature.
- Software qualities.
  - System requirements.
  - Where developers should focus?

What are the critical aspects in flight sw. design?

- Generic; mission-agnostic.
- Applicable both in educational and industry contexts.
- Our contribution to address critical aspects.

Guidelines for nano-satellite software design.

Illustrated with an software architecture.

- The 3Cat-1 software architecture.
→ Robustness through hierarchy:

- Boosts robustness from a purely architectural standpoint.
- Abstract/generic approach: logical ordering of components.
- Based on encapsulation and goal-oriented decomposition of functionalities.

- Abstraction layers: top ones are implement high-level functionality and do not rely on hardware (devices, buses, subsystems.)
- Each layer interacts with its adjacent levels:
  - Decompose (encapsulate) actions: commands, tasks, routines, functions…
  - Hierarchical relationships through robust communication channels (e.g. IPC, ITC…)
  - If modules are sufficiently isolated: removal of error propagation paths.
Design guidelines for nano-satellite flight software

→ Robustness through hierarchy:

- Horizontal fragmentation based on functionality.
- Functionalities are disseminated across levels of abstraction.
  - Minimizes complexity of each component.
  - From high-level (very abstract) goals to low-level (close to hardware) goals.
  - High-level (critical) are detached from hardware sources of error. E.g.:
    - Subsystem’s power failure.
    - File system failure.
    - Payload failure.

Illustrative goals

Abstraction

Totally detached from hw. \( \alpha_1 \)

functionality are much more intertwined

Prevent error propagation

Utterly hw. dependent

\[ \begin{align*}
\text{Prevent error propagation} \\
\text{Totally detached from hw.} \\
\text{functionality are much more intertwined} \\
\text{Utterly hw. dependent}
\end{align*} \]

\[ \begin{align*}
\alpha_1 & \quad \alpha_2 & \quad \alpha_3 & \quad \alpha_4 \\
\text{Prevent error propagation} & \quad \text{Totally detached from hw.} & \quad \text{functionality are much more intertwined} & \quad \text{Utterly hw. dependent}
\end{align*} \]

Illustrative goals

Functionality

- \( f_1 \) (e.g. energy management)
- \( f_2 \) (e.g. attitude control)
- \( f_3 \) (e.g. communications)
- \( f_4 \) (e.g. payload management)
- \( f_5 \) (e.g. data processing and storage)

Start high-level tasks, set mission constraints, trigger system states, validate sensor data with models.

Manage active processes/threads, expand states, decrypt telemetry commands, manage subsystems.

Send commands to devices, enable power regulators, log scientific data.
→ **Payload-oriented modularity:**

- Identifying the subsets which maximize internal coupling and minimize coupling between modules is a demanding intellectual exercise influenced by subjectivity.

- To achieve reusability, maintainability and flexibility: proper modularization should:
  - Identify what parts of the system are likely to change.
  - Locate those parts (i.e. their functionality) in specialized components.
  - Design inter-module interfaces that are insensitive to the anticipated changes, preventing the changeable aspects to be revealed by the interface.

- Changeable parts:
  - Subsystems: they are not removed, but can change (EPS, ADCS, COM…)
  - Payloads hosted by the spacecraft will differ from one mission to another.
→ **Payload-oriented modularity:**

- **Common interface for all low-level modules** (subsystem and payload control.)
- **Interface**: set of functions that can be invoked outside the module, namely:
  - `check()` – Verify that the subsystem does not present any error. Runs unit tests and reports issues.
  - `init()` – Cleanly initializes subsystem/payload and acquires static resources (e.g. memory, DB connection, starts peripheral’s drivers...)
  - `run()` – Executes routines.
  - `halt()` – Resets all system variables and devices, releases resources and exits.
  - **One-shot functions** – Interrupts or duplicates execution thread, to retrieve instantaneous data (e.g. sensors), trigger internal state transition or to perform one-time actions (e.g. enable DC/DC.)
    - The list of OSF is custom to each module.
Towards autonomous spacecraft:

- Providing autonomous mission planning capabilities.
- Concept initially explored by NASA’S DS-1 RAX.
- Autonomy system:
  - Ability to intelligently *plan* activities.
  - Ability to robustly *execute* this plan.
  - Based on mission goals (e.g. study crop evolution, capture images of volcano eruptions), deterministic environmental conditions (e.g. orbit position, input power) and system constraints (e.g. battery state-of-charge.)
Towards autonomous spacecraft:

- Architectural approach: 3 essential components to design an Autonomy System.
  - Task Planner:
    - Collects high-level goals → tasks.
    - Schedules system resource allocation for their execution (e.g. memory, storage, power.)
    - Could prioritize tasks.
  - Executive System:
    - Decomposes plan of action.
    - Perform all procedures to achieve it.
    - Monitors constraints are not violated.
  - Task Generation Engine:
    - Advanced/optional component.
    - On-board instrument data analysis to trigger task generation.
    - Checks system state and proposes maintenance tasks (e.g. reaction wheels desaturation, database maintenance.)

- Inflicts severe computational burden (CPU time, memory, power).
- Forces parts of the Autonomy Sys. to be enabled intermittently (scheduler).
Nano-satellite system and software characteristics

What are the critical aspects in flight sw. design?

Guidelines for nano-satellite software design.

Illustrated with an software architecture.

✓ Extracted from successful missions and the literature.
✓ Software qualities.
✓ System requirements.
✓ Where developers should focus?
✓ Generic; mission-agnostic.
✓ Applicable both in educational and industry contexts.
✓ Our contribution to address critical aspects.
✓ The ³Cat-1 software architecture.
Applying design criteria: mission context

→ The CubeCAT (3Cat) program:
  - Nano-Satellite and Payload Laboratory at UPC BarcelonaTech.
  - 3Cat-1:
    - Technology demonstrator mission.
    - Explore the payload capacity of a 1U CubeSat.
    - 7 payloads
      - Visible low-resolution CMOS camera.
      - Geiger counter based on COTS module.
      - CellSat: silicon Solar Cells.
      - Graphene Transistor characterization.
      - Wireless Power Transfer experiment.
      - MEMS-based atomic oxygen detector.
      - Self-powered beacon based on energy-harvesting techniques.
    - OBC: AT91SAM9G20, ARM7 at 400 MHz, 64 MB.
  - Test campaign in its last stage.
  - Spacecraft delivery in December 2015, launch tentative date Jan-Mar 2016.
  - 3Cat-2, 3Cat-3 currently in development.
Applying design criteria: ³Cat-1 software architecture

→ OS:
  ▪ Xenomai 2.6.2.1 + Linux kernel 2.6.35.9 (RT hypervisor approach).
  ▪ Software architecture deployed as non-real-time processes, and real-time tasks.

→ Four-tiered architecture:
  ▪ Inter Process Communication based on RT-pipes, RT-heaps, regular pipes, files and SQLite3 databases.

RT kernel managed

SYSTEM CORE

RT pipes

SYSTEM DATA BUS

pipes

PROCESS MANAGER

pipes

HARDWARE-DEPENDENT MODULES

conf. files

payload output files

Sys. Logs & houskeeping DB’s

Heaps

High-level system state control (global FSM), Energy states, Autonomy System (Task Planner)

System Executive: state expansion, high-level task decomposition (into processes, routines, etc.)

Command forwarder system. Transparent interface between Procman and HWmod.

Low-level payload and subsystem controllers. Encapsulated with a general purpose interface.
Applying design criteria: 3Cat-1 software architecture

→ Hardware-dependent modules (HWmod):
  - Encapsulation of payload and subsystem functionality → 7 processes with the same interface.

→ System Data Bus (SDB):
  - A transparent, reliable and secure interface to low-level processes.
  - Ad-hoc protocol:
    - 34 commands (4 to control process FSM and 30 OSF)
    - 7 control frames (ACK, asynchronous notifications…)

→ Process Manager (Procman):
  - A robust multi-threaded executive.
  - Executes high-level actions. Decomposes mission tasks into a set of sequential actions.
  - Manages low-level process states.

→ System Core (Syscore):
  - Monitors system vitals/variables: state-of-charge, temperatures, power input, latch-up’s.
  - Implements high-level (system) Finite State Machine.
  - Encompasses a Fully elastic, priority-based, multi-resource task planner.
Current nano-satellite trends have not focused on software issues.

- Nano-satellite designs have explored many techniques and designs to improve the spacecraft functionality, and system-wide qualities.

Due to the current context, three design areas can be improved: robustness, modularity oriented to payloads, spacecraft autonomy.

- Some of these have been extensively explored for large satellites.
- Nano-satellites can be an essential element in complex architectures and also require the exploration of software design techniques and architectural approaches.

Three design guidelines are proposed:

- Generic: mission-agnostic, applicable in educational and industrial programs.
- Proper encapsulation and hierarchy of components.
- Re-usable architectures; payload/subsystem modularization & interface definition.
- Nano-satellites may also benefit from autonomous capabilities.
  - Change of paradigm: from command-based to goal-oriented.
  - An enabler for nano-satellite constellations.
  - Reduced observability and bandwidth requires intelligent instrument control.
  - Computational requirements are critical and will determine the availability of such systems.
Thanks for your attention

Carles Araguz – carles.araguz@upc.edu
Elisenda Bou-Balust – elisenda.bou@upc.edu
Eduard Alarcón – eduard.alarcon@upc.edu
Backup slides
Nano-satellite software techniques and designs

→ Process isolation and protected memory.
  ▪ UNIX process model instead of TSP kernels/middleware.

→ Real-Time Operating Systems:
  ▪ Critical to avoid priority inversion, deterministic execution of critical tasks…
  ▪ Instead of using industry renowned products (i.e. VxWorks, RTEMS, QNX, LynxOS, …) nano-satellite developers tend to prefer free/open-source alternatives.
    • Small-footprint: FreeRTOS, uC/COS-III…
    • Soft-real-time Linux: PREEMPT_RT, Xenomai, uCLinux.

→ FDIR methodology:
  ▪ E.g. ESA’s OPS-SAT CubeSat: dedicated FDIR computer that monitors each payload through modular controller.
  ▪ Despite complex FDIR systems requiring high computational capabilities, some nano-satellite programs have analyzed fault trees, and designed procedures to circumvent errors. E.g. TU Delft’s DelFFi (Bräuer 2015).
Nano-satellite software techniques and designs

→ De-embeddable core:
  - CalPoly’s 2nd Gen. Bus (Manyak, 2011)

→ Decentralized/distributed approaches:
  - AAUSAT3 software (Bønding 2008)

Diagram extracted from Manyak 2011, and adapted
Nano-satellite software techniques and designs

→ Dynamically-linked libraries.
  ▪ Reusable code/segmented software: easy to update/maintain.

→ Software redundancy:
  ▪ Data redundancy: triplicate critical data (e.g. config. params.) in EEPROM (Hishmeh 2009).
  ▪ Bootloader redundancy: triplicate kernel images in NAND.

→ Other techniques:
  ▪ Robust communications:
    • Prevent unreliable delivery of digital data (over digital buses, e.g. SPI, I²C, CAN…)
    • Error Detection and Correction techniques (EDAC): data integrity checks, Ack/Nack, Handshakes, timeouts…
  ▪ Hardware and software watchdogs:
    • Restart modules when unexpected deadlocks happen.
    • Process heartbeat (between components; also among different CPU’s)
  ▪ Robust programming.
    • Strict coding rules.
    • When applying industry standards for software reliability is too complex or unfeasible, adopt simpler alternatives: G. J. Holzmann “The power of 10: rules for developing safety-critical code” (NASA/JPL Laboratory for Reliable Software)
Requirements for next-generation nano-satellite software

→ Assessing the goodness of the software is fundamental to understand the system’s strengths and weaknesses.


- Quality attributes:
  - Non-functional requirements.
  - Intertwined with each other: reliability ↔ performance ↔ portability, …
  - *Orthogonal* to the software functionality.

- Each quality attribute should be weighted in the context of system-specific goals.

- **Subjective** assessment:
  - The list of attributes is diverse.
  - Units or numerical representation are usually not defined.

- Subjective assessment:
  - The list of attributes is diverse.
  - Units or numerical representation are usually not defined.
Applying design criteria: 3Cat-1 software architecture

→ Hardware-dependent modules (HWmod)

- Subsystem HWmod:
  - EPS → Controls power management board, performs housekeeping.
  - ACS → Interfaces attitude control and determination board.
  - Comms. → Implements comms. protocol and delivers telemetry commands to the system.

- Payload HWmod:
  - MEMS, Camera, Geiger, WPT-GT → perform experiment/demonstration by interacting with the dedicated board and devices.
  - All of them implement a common interface (check(), init(), run(), halt() and 22 one-shot functions.)
  - Each process is configured with a set of PARAM=<value> pairs defined in their configuration files.
  - Processes start and remain in idle until a command arrives. If an error occurs, they are automatically restarted by the architecture.
Applying design criteria: ³Cat-1 software architecture

→ System Data Bus (SDB)
  - Transparently, reliably and securely forward commands and data (accounting for command permission level).
  - Encapsulates/hides low-level architectural structure and components.

- Implements a custom protocol:
  - 34 commands (4 to control process FSM and 30 OSF)
  - 7 control signals (ACK, asynchronous notifications…)

- SDB v2
  - Modular Command System → parametrized definition of mission commands: fixed at compile-time.
  - SDB is a common command interface for all architectural levels.
  - Multi-Level Security domains can be defined to protect critical/system commands.
Applying design criteria: \(^3\text{Cat-1} \) software architecture

→ Process Manager (Procman)
  - A robust multi-threaded executive.
  - Executes high-level actions.
    - Runs system states triggered by the Syscore.
    - Notifies subsystem failures (resulting from `HWmod.check()` invocations or from asynchronous signals).
  - Decomposes mission tasks into a set of sequential actions.
    - Defines and executes task handler routines.
    - Plan of action defined by Syscore in the schedule file.
  - Manages low-level process states.
    - Through the SDB protocol.
    - Writes/checks module parameters to configuration files.
  - Interacts with the kernel to control OBC power states.

Oct. 27th, 2015
Applying design criteria: ³Cat-1 software architecture

→ System Core (Syscore)
  - Critical management section (RT).
  - Monitors system vitals/variables: state-of-charge, temperatures, power input, latch-up’s.
  - Encompasses an on-board task planner.
  - System State Control:

Oct. 27th, 2015

General-purpose Payload-oriented Software Architecture for Nano-Satellites
2015 Flight Software Workshop
Applying design criteria: ³Cat-1 software architecture

→ Autonomy System: Task Planner
  ▪ Fully elastic, priority-based, multi-resource task planner.
  ▪ Constraint-Satisfaction Problem solver.
  ▪ Fully-elasticity:
    • Resources capacity: dynamic.
    • Task “consumptions”: variable.
  ▪ Types of dynamic resources:
    • *Instantaneous*, the removal of activities returns its capacity to the initial value (e.g. power, subsystem availability);
    • and *cumulative*, the removal of activities does not imply returning them to their initial capacity (e.g. energy, storage).

  ▪ Static constraints: environmental/mission constraints (predicted temperature, orbit position, comms. window...)
  ▪ Implemented in Prolog.
Applying design criteria: ³Cat-1 software architecture

→ Prolog Task Planner open problems:
  - Cumulative resource with intrinsic limits.
  - Final resource capacity.
  - **Performance** (CPU time and memory)

→ Affect problem complexity (worsens performance):
  - Number of time units (resolution, scheduling window width)
  - Number of tasks.
  - Number of resources consumed by each task.
  - Task consumption profiles.
  - Number of cumulative resources.
  - Context (inherent problem restrictions: too much constrained)


