



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

pyfgc and the FGC4 Software Stack: Redesign of the Python Communi- cation Library for the LHC

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA IN-
FORMATICA

Author: **Lorenzo Rossi**

Student ID: 995311

Advisor: Prof. Elisabetta di Nitto

Co-advisors: Przemyslaw Plutecki

Academic Year: 2025-2026

Abstract

The fourth generation of the Function Generator/Controller (FGC4) introduces a new communication protocol at CERN's power converter control infrastructure: the FGC Front-End Protocol (FFP), a JSON-over-WebSocket protocol that replaces the Network Command/Response Protocol (NCRP) used by previous generations. FGC4 is planned for deployment during Run 4 of the LHC and, in the longer term, in the proposed Future Circular Collider (FCC). Enabling control software to interact with FGC4 devices requires new protocol and transport support at every layer of the software stack.

This thesis presents the design and implementation of the software stack required to communicate with FGC4 devices via FFP. The primary contribution is `pyfgc 3.0.0`, a complete rewrite of the Python library for FGC device communication, introducing WebSocket transport and FFP support while retaining backward compatibility with FGC2 and FGC3 via NCRP over TCP/IP. The library separates the transport layer from the message serialization layer through generic abstractions, and a `SyncWrapper` utility generates all synchronous APIs from their asynchronous counterparts, eliminating code duplication. The companion packages `pyfgc_name`, `pyfgc_rbac`, and `pyfgc_statussrv` were rewritten alongside, replacing global mutable state with instance-based designs, introducing typed entity classes, and adopting modern Python type annotations throughout. The web-facing applications that depend on `pyfgc` were also updated: the FGC API, FGC Commander, FRESCO, and PowerSpy were migrated to the new library, and the FGC API was extensively reworked to adopt the `AutoConnectionFactory` pattern and a modern CI/CD pipeline. On the firmware side, the C++ `FfpClient` class was written from scratch for the FGCDv2 gateway, and three cross-cutting clean-up tasks were completed across 141 files covering error handling types and integer type aliases. CI/CD pipelines were introduced or extended for all packages, using the ACC-Py GitLab CI template library and `setuptools-scm` for automated versioning.

Validation was carried out at two levels. The unit test suite comprises 946 tests across six packages, all passing without hardware access, with branch coverage reaching 97% for `pyfgc`. Integration tests executed against real FGC4 and FGC3 devices at the CERN test bench confirmed correct end-to-end behaviour for both device generations. Robustness

tests verified stable operation under 100 concurrent requests and uninterrupted 24-hour sessions with no connection drops or memory growth.

Keywords: FGC4, FFP, WebSocket, Python, asyncio, CERN, LHC, power converters, protocol migration, CI/CD

Abstract in lingua italiana

La quarta generazione del Function Generator/Controller (FGC4) introduce un nuovo protocollo di comunicazione nell'infrastruttura di controllo dei convertitori di potenza al CERN: il FGC Front-End Protocol (FFP), un protocollo JSON su WebSocket che sostituisce il Network Command/Response Protocol (NCRP) utilizzato dalle generazioni precedenti. L'FGC4 è previsto per il Run 4 dell'LHC e, in prospettiva futura, per il Future Circular Collider (FCC). Per consentire al software di controllo di interagire con i dispositivi FGC4, è necessario supporto per il nuovo protocollo e trasporto a ogni livello dello stack software.

Questa tesi presenta la progettazione e l'implementazione dello stack software necessario per comunicare con i dispositivi FGC4 tramite FFP. Il contributo principale è `pyfgc 3.0.0`, una riscrittura completa della libreria Python per la comunicazione con i dispositivi FGC, che introduce il trasporto WebSocket e il supporto FFP mantenendo la retrocompatibilità con FGC2 e FGC3 tramite NCRP su TCP/IP. L'architettura della libreria separa il livello di trasporto dal livello di serializzazione dei messaggi tramite astrazioni generiche, e un'utilità `SyncWrapper` genera tutte le API sincrone a partire dalle controparti asincrone, eliminando la duplicazione del codice. I pacchetti complementari `pyfgc_name`, `pyfgc_rbac` e `pyfgc_statussrv` sono stati riscritti insieme alla libreria principale, sostituendo lo stato globale mutabile con design basati su istanze, introducendo classi tipizzate e adottando annotazioni di tipo Python moderne. Le applicazioni web che dipendono da `pyfgc` sono state aggiornate: FGC API, FGC Commander, FRESCO e PowerSpy sono stati migrati alla nuova libreria, e l'FGC API è stato ampiamente rielaborato con il pattern `AutoConnectionFactory` e una pipeline CI/CD moderna. Sul fronte firmware, la classe C++ `FfpClient` è stata sviluppata da zero per il gateway FGCDv2, e tre interventi trasversali sono stati completati su 141 file, riguardanti la gestione degli errori e gli alias di tipo intero. Pipeline CI/CD sono state introdotte o estese per tutti i pacchetti, utilizzando la libreria di template GitLab CI di ACC-Py e `setuptools-scm` per la versionatura automatica.

La validazione è stata condotta su due livelli. La suite di test unitari comprende 946 test su sei pacchetti, tutti superati senza accesso all'hardware, con una copertura dei

rami che raggiunge il 97% per `pyfgc`. I test di integrazione eseguiti su dispositivi FGC4 e FGC3 reali al banco di test del CERN hanno confermato il corretto comportamento end-to-end per entrambe le generazioni di dispositivi. I test di robustezza hanno verificato il corretto funzionamento sotto 100 richieste concorrenti e sessioni ininterrotte di 24 ore senza interruzioni di connessione o crescita della memoria.

Parole chiave: FGC4, FFP, WebSocket, Python, asyncio, CERN, LHC, convertitori di potenza, migrazione di protocollo, CI/CD

Contents

Abstract	i
Abstract in lingua italiana	iii
Contents	v
1 Introduction	1
1.1 Objectives	1
1.2 Scope of the thesis	4
1.3 Impacts of the contribution	5
2 Analysis of the state of the art	7
2.1 FGCs	7
2.1.1 The Large Hadron Collider	7
2.1.2 Power distribution for the accelerator complex	7
2.1.3 Function Generator Controllers	9
2.1.4 POPS and POPS+	12
2.1.5 SVC	14
2.2 Communication Protocols at CERN	14
2.2.1 Network architecture	14
2.2.2 CMW: Controls Middleware	15
2.2.3 CORBA, JMS, and OPC	16
2.2.4 TCP/IP-based protocols	17
2.3 Application Protocols at CERN	17
2.3.1 RDA	18
2.3.2 SCRP	19
2.3.3 NCRP	19
2.4 Authentication and Authorization: RBAC	21
2.5 FESA	23

2.6	Pre-existing software for FGCs	23
2.6.1	Firmware	23
2.6.2	Control Software	23
2.6.3	ACC-Py	27
2.6.4	Python client libraries	28
3	Problems	31
3.1	Problems with current solutions	31
3.1.1	Lack of FGC4 support	31
3.1.2	Implicitly synchronous communication	32
3.1.3	Global mutable state	32
3.1.4	Procedural API and implicit shared state	34
3.1.5	Absence of a structured RBAC token type	35
3.1.6	Code duplication between synchronous and asynchronous paths	36
3.1.7	Weak type safety and string-based dispatch	37
3.1.8	Legacy Python version and absence of code quality tooling	39
3.1.9	Absence of docstrings and documentation	40
3.1.10	Tightly coupled code	41
3.1.11	No high-level connection API	41
3.1.12	Absence of meaningful automated tests	43
4	Content of the thesis	45
4.1	Idea definition	45
4.2	pyfgc: architecture overview	46
4.3	Channel layer	47
4.4	Protocol layer	49
4.4.1	FFPPProtocol	49
4.4.2	NCRPPProtocol	50
4.4.3	SCRPPProtocol and CMWProtocol	50
4.5	Connection layer	50
4.6	AutoConnection layer	52
4.6.1	CMW/RDA subscription support	56
4.7	Real-time data	57
4.8	Utilities	57
4.9	Companion packages	58
4.9.1	pyfgc_name	58
4.9.2	pyfgc_rbac	59
4.9.3	pyfgc_statussrv	60

4.9.4	pyfgc_decoders	61
4.9.5	ccs-fastapi-utils	62
4.9.6	pyfgc_test_framework	62
4.10	Web application integration	63
4.10.1	FRESCO	63
4.10.2	FGC Commander	63
4.10.3	FGC API	63
4.10.4	PowerSpy	66
4.11	DevOps and CI/CD extensions for testing	66
4.11.1	Development methodology	66
4.11.2	Python package pipeline	67
4.11.3	Web application pipeline	68
4.11.4	Testing infrastructure	69
4.12	Contributions to the FGCDv2 gateway firmware	69
4.12.1	FFP client implementation (FGCD-122)	72
4.12.2	Error handling standardisation (FGCD-211)	73
4.12.3	Integer type standardisation (FGCD-212)	75
4.12.4	FFP protocol interface cleanup (FGCD-221)	75
4.13	Risks and trade-offs	76
5	Validation and results	77
5.1	Validation goals	77
5.2	Validation strategy	77
5.2.1	Validation environment	77
5.2.2	Testing strategy	78
5.2.3	Acceptance criteria	80
5.3	Validation results	80
5.3.1	Unit tests	80
5.3.2	Static analysis and code quality	81
5.3.3	Integration tests	82
5.3.4	Performance	84
5.3.5	Compatibility	84
5.3.6	Reliability and robustness	85
5.4	Validation summary	87
6	Conclusions and future developments	89
6.1	Summary of the results	89
6.2	Future developments	91

Bibliography	93
List of Figures	97
List of Tables	103
Listings	105
Ringraziamenti	107

1 | Introduction

1.1. Objectives

The work presented in this thesis was carried out at CERN, the European Organization for Nuclear Research (French: Organisation europeenne pour la recherche nucleaire), located on the border of Switzerland and France near Geneva. Founded in 1954, originally under the name Conseil Europeen pour la Recherche Nucleaire (from which the acronym CERN is derived), CERN is the world's largest laboratory for particle physics research, operating a complex of interconnected particle accelerators and detectors that culminate in the Large Hadron Collider (LHC). The organization employs approximately 2500 staff members and hosts thousands of users from research institutions worldwide, including more than 12000 visiting scientists and students from more than 80 universities and institutes around the world.

The author's internship was carried out within the SY-EPC-CCS section (Converter Controls Software), part of the SY-EPC (Electrical Power Converters) group in the SY (Systems) department. The CCS section is responsible for developing and operating the software infrastructure used to monitor and control the more than 5000 power converters distributed across CERN's accelerator complex [23]. It comprises approximately 15 to 20 engineers organized into three teams: Converter Control, Industrial Controls, and Tools & Services. The author worked within the Converter Control team, which is specifically responsible for the FGC (Function Generator/Controller) platform and its associated software ecosystem. This team develops and maintains the firmware, embedded control software, and client libraries that enable operator applications to communicate with and command the FGC devices governing current regulation in the accelerator magnets.

The fourth generation of the Function Generator/Controller (FGC4) introduces a new communication model based on the FGC Front-End Protocol (FFP), a JSON-over-WebSocket protocol that replaces the previously used Network Command/Response Protocol (NCRP) [24]. FGC4 devices are planned to enter operation during Run 4 of the LHC, which follows Long Shutdown 3 (LS3, from September 2026 to June 2030), the major



Figure 1.1: Aerial view of the CERN site near Geneva. The 27 km LHC ring is traced in yellow; the smaller SPS ring (7 km) and the CERN campuses are visible on the top right. (Image credit: CERN)

maintenance and upgrade period dedicated to the High-Luminosity LHC (HL-LHC) [4]. Beyond the HL-LHC, FGC4 is also expected to be adopted in the Future Circular Collider (FCC), a proposed successor to the LHC currently under feasibility study, which would reuse part of CERN’s existing accelerator complex as an injector chain [5]. Enabling control software to communicate with FGC4 devices therefore requires new protocol and transport support at every layer of the software stack.

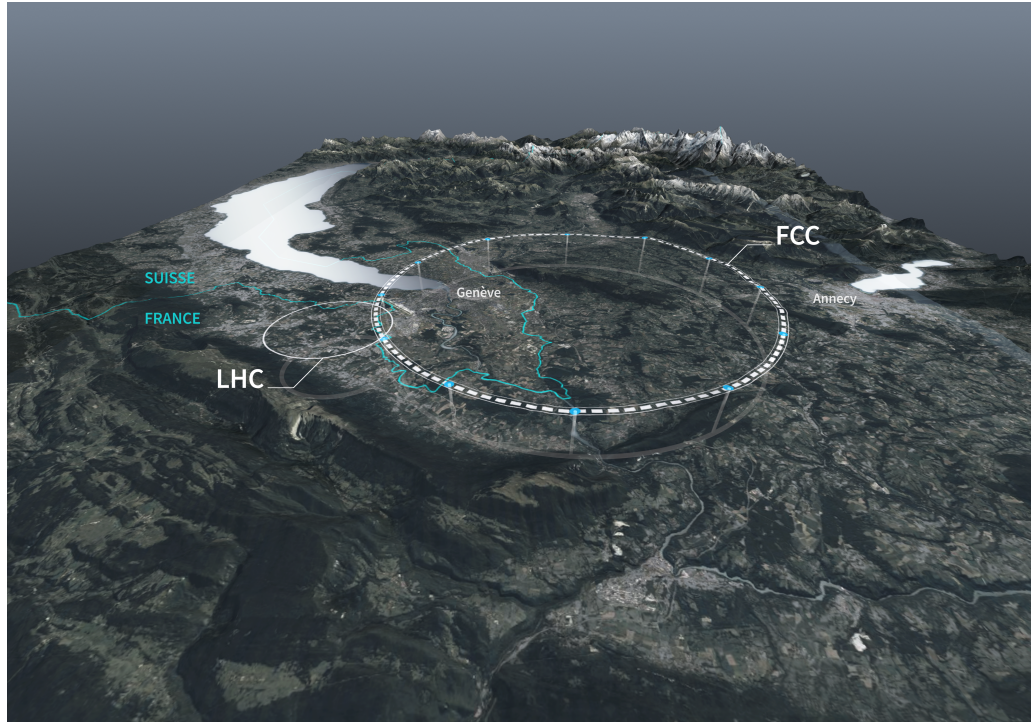


Figure 1.2: Artist’s rendering of the proposed 91 km FCC tunnel (dashed) overlaid on the existing HL-LHC (white), which would serve as part of its injector chain. (Image credit: CERN)

The primary objective of this thesis is the design and implementation of the Python software stack required to communicate with FGC4 devices via FFP. This software stack is centered on `pyfgc`, the Python client library used by control applications to communicate with FGC devices. This work encompasses the following contributions:

- **Analysis of the existing ecosystem.** Study of the limitations of the `pyfgc` 1.x and 2.x library ecosystem with respect to FGC4 requirements, covering both functional gaps and software quality deficiencies.
- **Study of the FFP protocol.** Analysis of the FFP specification to understand its message types, communication model, authentication mechanism, and differences from NCRP.

- **Rewrite of the pyfgc Python library ecosystem.** Complete redesign and implementation of the `pyfgc` library (version 3.0.0) and its companion packages (`pyfgc_name`, `pyfgc_rbac`, `pyfgc_statusrv`) to support FFP over WebSockets while retaining backward compatibility with FGC2 and FGC3 via NCRP over TCP/IP.
- **Contributions to the FGCDv2 (FGC Daemon v2) gateway firmware.** Implementation of the C++ `FfpClient` class, the firmware-side counterpart of the Python FFP client library; replacement of third-party error-handling and integer-type utilities with standard-library equivalents; and cleanup of the FFP protocol public interface headers.
- **Integration and modernization of existing FGC control software.** Update of the FGC API and PowerSpy to use the new `pyfgc` 3.0.0 library and FFP support. FGC Commander benefits indirectly through the FGC API migration. The current status of FRESCO and its planned migration path are also discussed.
- **Testing and validation.** Development of unit tests and integration tests for the new library, and validation of end-to-end communication with FGC4 devices.

1.2. Scope of the thesis

This thesis covers the software work carried out as part of the FGC4 control system integration project at CERN. The geographical scope is limited to CERN's facilities in Geneva, Switzerland and Prévessin, France, and the surrounding area, where the FGC4 hardware and test infrastructure are located. The work does not address the hardware design of FGC4 or the firmware below the FGCDv2 layer; both are outside the scope of this thesis and are treated as pre-existing components.

The contributions span four areas. First, the `pyfgc` 3.0.0 Python ecosystem provides the communication layer between control applications and FGC devices; it is the primary deliverable of this thesis. Second, contributions to the FGCDv2 gateway firmware written in C++ extend the scope to the server side of the communication stack: the `FfpClient` class was implemented from scratch, third-party error-handling and integer-type utilities were replaced with standard-library equivalents across the codebase, and the FFP protocol public interface headers were cleaned up. Third, the web-facing control applications built on top of `pyfgc` were updated to support FGC4: the FGC API service was refactored and extended, and PowerSpy was migrated to the new library. Fourth, DevOps and CI/CD extensions were introduced for both the Python monorepo and the web tools repository,

covering automated test pipelines, containerised deployments, and version management infrastructure.

The work was carried out primarily in Python 3.14 using `asyncio` and the `websockets` library. The C++ contributions target the FGCDv2 framework.

1.3. Impacts of the contribution

The work presented in this thesis delivers the communication layer that enables FGC4 devices to be integrated into CERN’s power converter control infrastructure. Prior to this work, no control application could interact with FGC4 hardware: the `pyfgc` ecosystem lacked both WebSocket transport and FFP protocol support, making the FGC4 integration project impossible to proceed without a new software communication layer.

The primary impact is the `pyfgc 3.0.0` ecosystem, which for the first time allows control software to communicate with FGC4 devices via FFP over WebSockets, while retaining full backward compatibility with the more than 1500 FGC3 devices already in operation. The FGC API and PowerSpy services, which are the main programmatic interfaces used by control applications and operators at CERN, have been updated to use the new library and now support FGC4 natively. FGC Commander benefits indirectly through the FGC API migration.

Beyond the functional enablement, the rewrite addresses a set of structural quality deficiencies in the previous library that would have made it difficult to maintain and extend the communication stack as the FGC4 deployment scales up across the LHC and, in the longer term, the FCC. The new architecture cleanly separates transport from protocol, eliminates the code duplication that required every feature to be implemented twice, and replaces global mutable state with instance-based designs throughout the companion packages. A comprehensive test suite of 946 unit tests, together with integration tests against real FGC4 and FGC3 hardware, provides confidence in the correctness and robustness of the implementation. These structural improvements are described in chapter 3; the design decisions that address them are discussed in chapter 4.

2 | Analysis of the state of the art

2.1. FGCs

As introduced in Chapter 1, CERN’s research activity is carried out through the use of particle accelerators, organized in the so-called *accelerator complex*, where they are interconnected for operation. The complex consists of a chain of accelerators operating mainly with protons: starting from the Linear Accelerator 4 (LINAC4), protons are injected at 160 MeV into the Proton Synchrotron Booster (PSB), then accelerated through the Proton Synchrotron (PS) and the Super Proton Synchrotron (SPS), and finally into the Large Hadron Collider (LHC). The complex operates 24 hours a day, seven days a week for most of the year, imposing strict requirements of high reliability and availability on all its subsystems.

2.1.1. The Large Hadron Collider

The LHC is a superconducting ring-shaped particle accelerator, the largest in the world, with a circumference of 27 km [16]. Located underground at a depth of approximately 100 m, it circulates two particle beams in opposite directions within two separate beam pipes. The beams are brought together at four interaction points, where the four main experiments are located: A Toroidal LHC Apparatus (ATLAS), Compact Muon Solenoid (CMS), A Large Ion Collider Experiment (ALICE), and Large Hadron Collider beauty (LHCb). The ATLAS and CMS experiments confirmed the existence of the Higgs boson in 2012 [1, 11].

2.1.2. Power distribution for the accelerator complex

Particle accelerators are constituted by magnets, which bend and shape the beam, and by RF cavities, which accelerate it. The magnets are powered by power converters, which convert AC power from the electrical grid into precisely controlled DC power. In the LHC, dipole magnets keep the beam in orbit using superconducting coils cooled to 1.9 K, through which a current in the order of 10 kA is passed to produce a magnetic field of

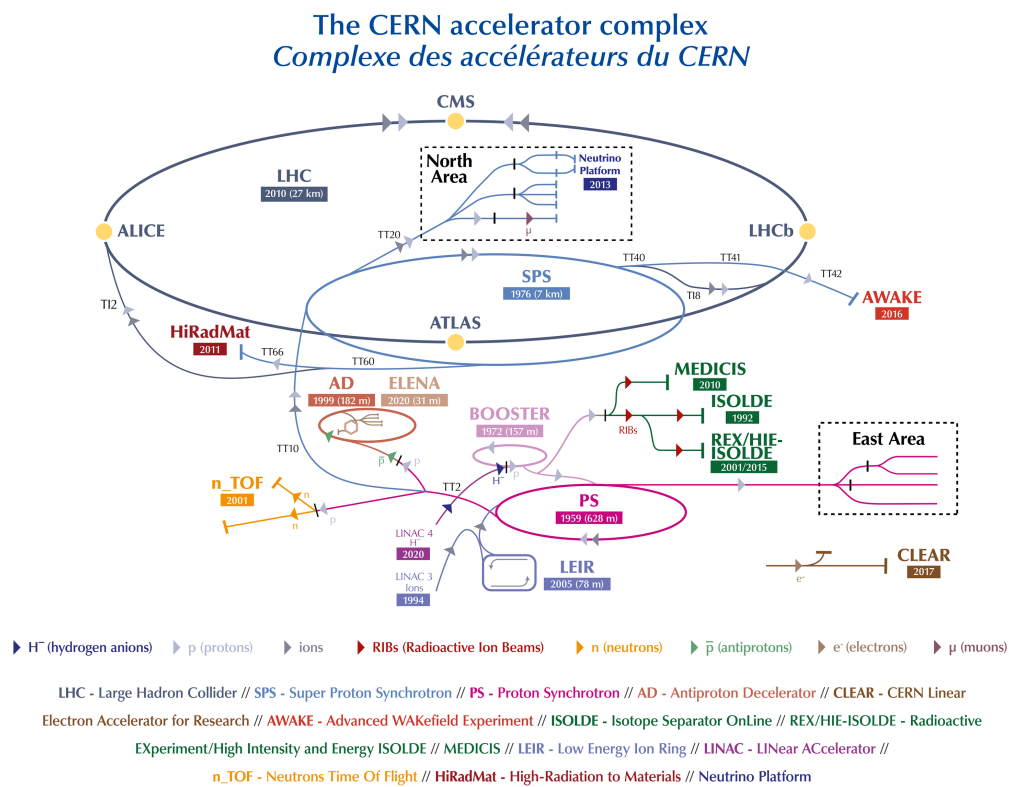


Figure 2.1: The CERN accelerator complex. Particles are progressively accelerated through a chain of machines before being injected into the LHC. (Image credit: CERN)

approximately 8 T [16]. Since the bending radius of the beam is directly proportional to the current, precise and stable current control is critical for correct operation.

In CERN's installations, the current control loop is implemented in a modular platform called Regulating FGC (RegFGC), which hosts the Function Generator Controller (FGC). The FGC is responsible for generating the current reference waveform and closing the feedback control loop by reading back the output current through precision sensors such as Direct-Current Current Transformers (DCCTs).

2.1.3. Function Generator Controllers

The Function Generator/Controller (FGC) is a programmable embedded controller that implements the current control algorithm for CERN's power converters. Since its introduction, the FGC platform has evolved through multiple generations to meet the growing demands of the accelerator complex in terms of performance, scalability, and reliability.

FGC2

The second generation of the Function Generator/Controller (FGC2) was developed between 1998 and 2005. Designed to serve the 1700 power converters in the LHC and its experiments, the FGC2 has been in continuous operation in the LHC since 2008 [10].



Figure 2.2: The FGC2 hardware module. (Image credit: CERN)

FGCLite

Due to radiation-induced beam dumps (a safety mechanism that extracts and disposes of the circulating beam when an anomaly is detected) encountered in the LHC tunnel, a radiation-tolerant variant of the FGC2 platform known as FGCLite was developed between 2013 and 2016. Of the original 1700 units, 710 FGC2 devices remain in operation today, alongside 1022 FGCLite devices that have taken over the rest [10].

FGC3

The third generation of the Function Generator/Controller (FGC3) was developed between 2007 and 2011, and is currently the most widely deployed platform at CERN. Over 1500 FGC3 devices are in operation controlling a large variety of power converters across all accelerators and experimental areas. The FGC3 embedded hardware comprises a microprocessor, a DSP, a microcontroller, and an FPGA used to implement peripherals and glue logic [10].



Figure 2.3: The FGC3 hardware module. (Image credit: CERN)

FGC4

The fourth generation of the Function Generator/Controller (FGC4) is currently in active development. It will use the DI/OT (Distributed I/O Tier) platform as its hardware base [14]. DI/OT is a modular, FPGA-based hardware platform developed at CERN for accelerator control, built around a 3U crate compliant with the CompactPCI Serial industrial standard. The platform provides both radiation-tolerant and high-performance variants of its system boards, supports a variety of communication mezzanines, and is designed to be shared across equipment groups through application-specific peripherals and FPGA configurations. Both the DI/OT hardware and the FGC4 controller represent

entirely new products (in terms of hardware as well as software), requiring a substantial number of design decisions during the development phase [10].

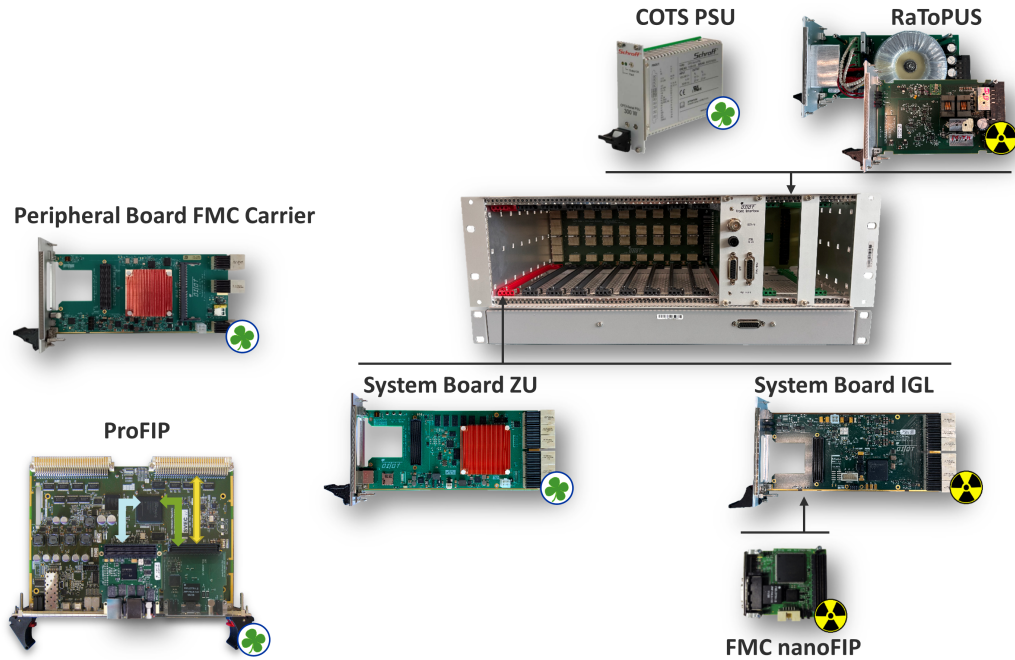


Figure 2.4: The DI/OT hardware kit. The 3U CompactPCI Serial crate (centre) accommodates the System Board ZU (Zynq UltraScale+ MPSoC, non-radiation-tolerant) and the System Board IGL (Igloo2 FPGA, radiation-tolerant, marked with the radiation symbol). Power is supplied by either a COTS PSU or the radiation-tolerant RaToPUS. Application-specific Peripheral Boards (shown left) plug into the backplane. (Image credit: CERN / Open Hardware Repository [14])



Figure 2.5: The FGC4 prototype board. (Image credit: CERN)

Device classes

Each device in the FGC ecosystem is assigned a numeric class identifier that governs its behaviour, available properties, and network access method. Two distinct class hierarchies coexist. The first covers the FGC firmware itself: each class defines the control algorithm variant and the set of properties exposed to the control system, corresponding to a specific

application type within a given hardware generation (e.g., a standard power converter or a static var compensator). The property sets exposed by each firmware class are not interchangeable: a property available on one class (e.g. `MEAS.SVC.I.FEEDER.R` on `FGC4_SVC`) may not exist on another (e.g. `FGC4_PC`), and conversely two classes may expose properties with the same name but different types, ranges, or semantics. The full, auto-generated reference of properties and configuration properties for each class is published online [25]. The second hierarchy covers the FGC Daemon (FGCD), the software process running on the front-end computer that acts as the network gateway between the control system and the physical FGC hardware. The gateway class determines which network protocol is used to communicate with the attached devices: gateways of class `FGCD_GW_ETHER` serve FGC2 and FGC3 devices over NCRP, while gateways of class `FGCD_GW_FGC4` serve FGC4 devices over FFP on WebSockets. FGC4 is currently the only platform for which FFP over WebSockets is supported. The class identifier of the gateway is recorded in the *name file* for each addressable device. A name file is a plain-text, colon-separated configuration file that maps each device name to its gateway host address, fieldbus address, class identifier, and operational mode mask; it is created by the SY-EPC-CCS section and maintained, along with other accelerator configuration data, in a dedicated GitLab repository [27], from which it is also mirrored to an NFS share and an HTTP server for client access. This record allows the software stack to select the appropriate protocol automatically without caller intervention, as described in Chapter 4.

Table 2.1: Principal FGC firmware device classes.

Class ID	Name	Platform	Description
51	FGC2_LHC_PC	FGC2	LHC power converter
53	FGC2_POPS	FGC2	POPS converter
59	FGC2_RF_FGEN	FGC2	RF function generator
60	FGC3_BASE	FGC3	FGC3 boot class
62	FGC3_FPC	FGC3	Fast pulsed power converter
63	FGC3_PC	FGC3	Power converter
64	FGC3_SVC	FGC3	Static var compensator
65	FGC3_EPIC	FGC3	EPIC converter
71	FGC4_PC	FGC4	Power converter
72	FGC4_SVC	FGC4	Static var compensator

2.1.4. POPS and POPS+

The main power supply of the Proton Synchrotron (PS) accelerator is one of the largest at CERN. The original rotating machine system, in operation since 1968, was replaced

Table 2.2: Principal FGCD gateway classes and the protocol each uses.

Class ID	Name	Protocol	Notes
1	FGCD_GW_WFIP	NCRP	WorldFIP gateway (FGC2)
2	FGCD_STATUS	NCRP	Status server gateway
4	FGCD_GW_ETHER	NCRP	Ethernet gateway (FGC2/FGC3)
5	FGCD_GW_SERIAL	SCRP	Serial gateway
6	FGCD_GW_FGCLITE	NCRP	FGCLite WorldFIP gateway
21	FGCD_GW_FGC4	FFP	FGC4 gateway (WebSockets)
98	FGCD_EQ_STATUS	NCRP	Status server device

with a new NPC-based DC/DC power converter named POPS (POWer for PS), which uses capacitor banks as an energy storage medium [2, 20]. The system delivers a power of 60 MW and supplies a current of up to 6000 A to the PS main magnets, producing a magnetic field exceeding 1.2 T with a precision of approximately 0.01 mT. The power electronics are housed in six containers, each holding 60 tonnes of capacitors and eight power converters. Within each accelerator cycle, POPS delivers a high-power pulse to the magnets and reabsorbs the energy in approximately two seconds. POPS was connected to the 101 main magnets of the PS on 31 January 2011, and the first beam circulated under its control on 11 February 2011 [9].

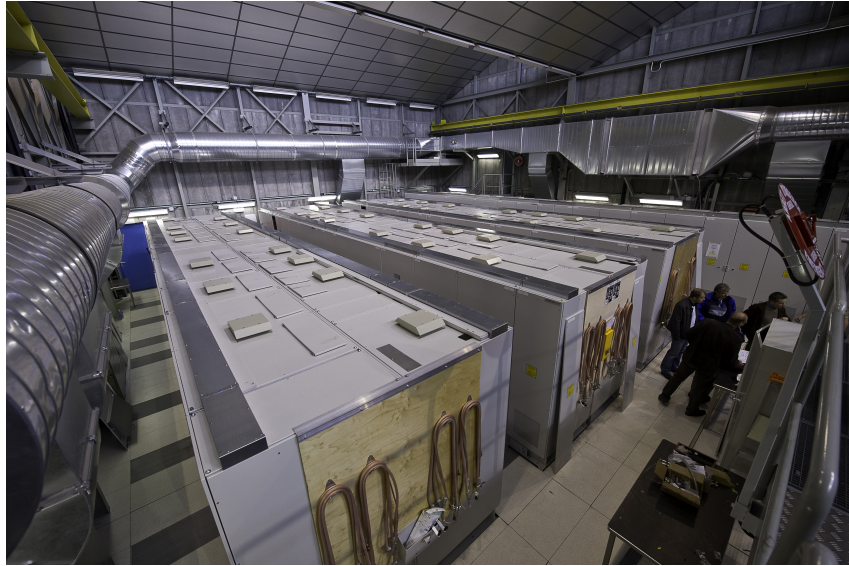


Figure 2.6: The POPS power converter system. (Image credit: CERN)

POPS+ is the planned upgrade of the POPS system. Unlike its predecessor, POPS+ will be controlled by an FGC4, making it one of the first deployments of the new controller generation in a high-power converter application.

2.1.5. SVC

A Static Var Compensator (SVC) is a device used to compensate the reactive power consumed by a load in an electrical power system. Due to its fast response, an SVC can stabilize busbar voltage even during rapid load fluctuations. SVCs employ reactive elements (typically thyristor-controlled reactors and capacitor banks) whose output is adjusted continuously to counteract the reactive power swings of the connected load. At CERN, SVCs also provide harmonic filtering of the distortion generated by the twelve-pulse thyristor power converters feeding the main accelerator magnets [18].

The SPS pulsed electrical network, which powers the SPS dipoles, quadrupoles, RF cavities, and the north experimental area, requires SVCs to manage the large reactive power swings caused by its pulsed loads. Without SVCs, the 18 kV busbar of the SPS pulsed network would experience periodic voltage variations of approximately 14% during each machine cycle; with SVCs in operation, these variations are reduced to below 0.75% [17].

The SVC units at CERN were candidates for migration to the FGC4 control platform. However, following a hardware failure in one of the SVC units, the integration project was suspended pending restoration of the full SVC installation. As a result, SVC control is excluded from the scope of this thesis.

This hardware failure occurred shortly after the LHC reached its 2024 integrated luminosity target of 110 fb^{-1} , when one of the LHC's SVC units unexpectedly switched off. During the procedure to reactivate it, an electrical disturbance damaged one of the two cryogenic compressor stations at LHC Point 8, which cools a section of the LHC arc operating at 1.9 K. The resulting loss of cooling capacity would have required reducing the number of circulating bunches from 2352 to approximately 1500 per beam, significantly impacting luminosity production [3].

2.2. Communication Protocols at CERN

2.2.1. Network architecture

CERN operates two logically separate networks. The Technical Network (TN) interconnects equipment essential for the safe operation of the accelerators (front-end computers, power converter controllers, and accelerator instrumentation) and connects this equipment to other CERN networks via a protective gateway [19]. The General Purpose Network (GPN) provides standard network connectivity for non-operational systems, including developer workstations and administrative computers. A set of dual-homed nodes designated as



Figure 2.7: The SVC installation affected by the hardware failure. (Image credit: CERN)

TN-trusted act as a bridge between the two networks, allowing controlled communication between GPN hosts and TN devices while themselves having no internet access.

This topology has direct implications for software that must communicate with FGC devices: client applications running on developer machines reside on the GPN, while the FGC controllers reside on the TN. Any connection must therefore route through a TN-trusted node, and network libraries must account for this constraint when establishing connections. Figure 2.8 illustrates this arrangement.

2.2.2. CMW: Controls Middleware

The Controls Middleware (CMW) is a communication framework developed at CERN to facilitate interaction between the various components of its control systems. It provides a unified middleware layer used to operate the CERN accelerator complex.

The remote procedure call system that preceded CMW, introduced in 1988, served CERN's control systems for decades using the RPC paradigm with equipment access standards. However, this stack had several limitations that made it unsuitable for the LHC era, including difficulty integrating industrial components and the inherent complexity of its architecture [31].

To address these limitations, CERN launched the Controls Middleware (CMW) project in 1999 to capture requirements and provide a more suitable communication infrastructure [15].

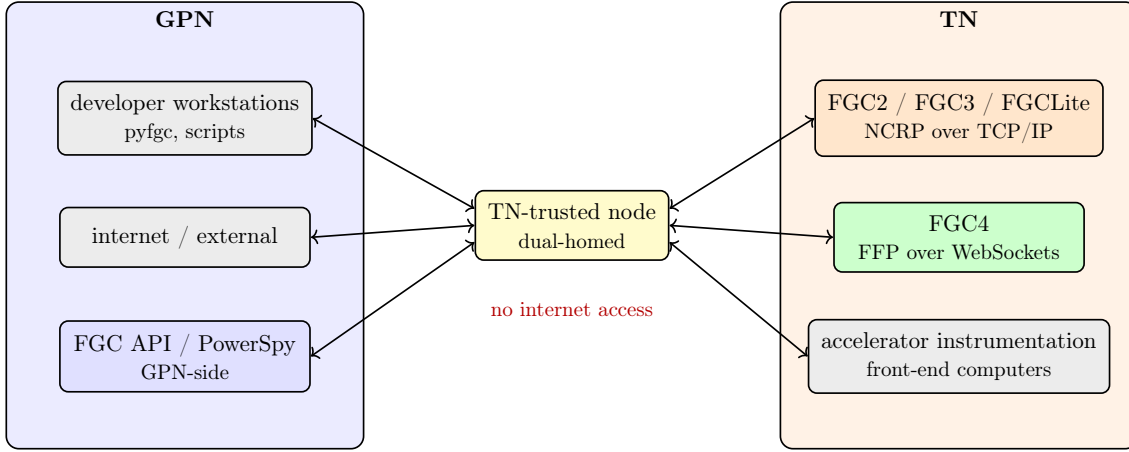


Figure 2.8: CERN network topology. The GPN hosts developer workstations and web-facing applications; the TN hosts FGC controllers and accelerator instrumentation. TN-trusted nodes are dual-homed machines that bridge the two networks while having no internet access themselves.

Currently, two main communication stacks are used at CERN: one based on CORBA, JMS, and OPC, and another consisting of various TCP/IP-based protocols. Each stack has its own specific use cases and characteristics, described in the following sections.

2.2.3. CORBA, JMS, and OPC

CORBA CORBA (Common Object Request Broker Architecture) is a standard for building distributed systems, defined by the Object Management Group (OMG). Its core feature is enabling transparent communication between objects implemented in different programming languages and running on different platforms, through a common object model and standardized interfaces.

CORBA features a language-neutral interface description language called IDL (Interface Definition Language), which allows developers to specify the interfaces of distributed objects in a platform-agnostic way. These interface definitions are compiled by code generation tools into language-specific stubs (e.g., for C++, Java, or Python), which handle communication transparently.

The central component of CORBA is the Object Request Broker (ORB), which manages the communication between clients and servers. Clients look up object references in a naming service and invoke methods on them; the ORB handles marshalling, transport, and unmarshalling transparently over the standardized IIOP (Internet Inter-ORB Protocol), ensuring interoperability between different CORBA implementations. Despite its wide

adoption in the 1990s for enterprise distributed systems, CORBA has seen declining use due to its complexity and the rise of simpler alternatives.

JMS JMS (Jakarta Messaging, previously Java Message Service) is a Java API for message-oriented middleware, allowing Java applications to exchange messages in a loosely coupled and often asynchronous manner. A JMS provider (or message broker) implements the API and manages message delivery; JMS clients interact with any compliant provider through common interfaces.

JMS supports two primary messaging models. In the *Point-to-Point* model, messages are sent to a specific queue and consumed by a single receiver, making it suitable for task distribution. In the *Publish/Subscribe* model, messages are published to a topic and delivered to all active subscribers, making it suitable for event broadcast.

OPC OPC (Open Platform Communications) is a family of standards used in industrial automation to exchange data between field devices (PLCs, sensors, controllers) and higher-level control systems (SCADA, HMI). Originally based on Microsoft's COM/DCOM technology, OPC has evolved into a vendor-neutral standard that enables interoperability across different hardware and software components in industrial environments. At CERN, OPC is used as a gateway between the CMW-based control system and industrial equipment [15].

2.2.4. TCP/IP-based protocols

In addition to the CORBA-based stack, CERN employs various TCP/IP-based protocols for direct communication with front-end computers and embedded controllers. These protocols are more lightweight than CORBA and require no ORB infrastructure. However, the TCP/IP-based protocols traditionally used at CERN followed an implicitly synchronous request/response paradigm, lacked native publish/subscribe mechanisms, and were not designed for access from web browsers or modern scripting environments. As the CERN software ecosystem has evolved, these limitations have driven the development of new front-end protocols based on WebSockets, which provide persistent full-duplex connections and are natively accessible from web environments [15, 26].

2.3. Application Protocols at CERN

Three application-level protocols are used for communication with FGC front-end controllers at CERN: RDA, SCRP, and NCRP.

2.3.1. RDA

Remote Device Access (RDA) is a software package developed at CERN as part of the CMW project [31]. It implements the *Accelerator Device Model*, in which physical devices are represented as named entities accessible through *properties*. Client applications can read, write, and subscribe to device property values through a unified API, regardless of where the device server is running in the network.

RDA is built on a client-server architecture. Device servers host and expose property values; client applications access them using the RDA client library. The underlying transport is CORBA-based, but this is hidden from the user: all CORBA details are internal to the RDA package and do not appear in its public API. This design makes RDA multi-language and multi-platform, with implementations in Java, C++, and Python. In the CERN infrastructure, a dedicated CMW proxy process bridges RDA clients and device servers, forwarding requests and responses between the two parties [21].

Devices are addressed by symbolic name (e.g. `RFNA.866.01.ETH3`) and properties by name (e.g. `MEAS.I`). Properties of FESA-class devices may expose individual sub-fields, accessible by field name alongside the property name. An optional cycle selector restricts a query to the data acquired during a specific timing cycle (e.g. `PSB.USER.MD1`), which is important for FGC devices whose measurements and reference waveforms are naturally cycle-specific. All RDA operations require RBAC authentication, which can be performed in explicit mode (username and password on every call), location-based mode (for development and test machines where the user’s location grants access automatically), or disabled mode (for servers where RBAC has been explicitly turned off) [21].

RDA supports both synchronous and asynchronous invocations, as well as a publish/subscribe mechanism for monitoring property values. Automatic connection monitoring and recovery are handled transparently by the framework. Standard command-line tools (`rda-get`, `rda-set`, `rda-subscribe`) are provided for interactive and scripted use and serve as the primary diagnostic interface for operator and developer access to devices over CMW [21].

Within the CERN control system, RDA3 (the current version) serves as the interface between high-level operator applications and device servers on front-end computers. As noted in the FFP specification [24], RDA3 “is used to communicate with the CERN control system”, as opposed to front-end-specific protocols such as FFP, which target direct communication between client software and the FGC4 devices themselves.

2.3.2. SCRP

The Serial Command/Response Protocol (SCRP) is a serial protocol for direct communication with FGC2 and FGC3 devices, using text-based commands over a local serial interface [26]. FGC2 exposes an RS232 port operating at 9600 baud (8 bits, no parity, 1 stop bit); FGC3 exposes a USB interface. Both configurations support Xon/Xoff flow control to throttle character emission by the FGC.

The protocol defines exactly two commands: set (!S) and get (!G). Each command identifies a property by name and is terminated by a newline (0x0A), a carriage return (0x0D), or a semicolon (0x3B). An optional PPM index in the range 1 to 32, written as PROPERTY(PPM), selects a specific function-generator programme. Array addressing supports four forms: a single element ([idx]), an open-ended range from a starting index ([from,]), a closed range ([from,to]), and a step range ([from,to,step]). Responses are strictly synchronous: the FGC returns exactly one response per command. A successful set response is the bare delimiter \$ followed by a newline; a successful get response prepends the property data. Error responses carry an error number and a short message and are terminated by \n! rather than \n;.

The serial port operates in one of three modes. *Direct mode* (the default after reset) is intended for control software: received characters are not echoed and no line editor is active. *Editor mode*, activated by sending ESC twice, provides an ANSI/VT100 line editor with a 16-entry command history, suitable for interactive use by human operators via a terminal emulator. A third mode, *Diagnostic mode*, streams binary real-time diagnostic data; it is deprecated and unsupported on FGC3.

Due to its serial nature, SCRP is limited in bandwidth and is not suited to networked control scenarios. Its use is therefore restricted to local, on-site access for debugging and commissioning tasks.

2.3.3. NCRP

The Network Command/Response Protocol (NCRP) is an ASCII protocol carried over TCP/IP for networked communication with FGC front-end computers [26]. Establishing a connection to an FGC gateway requires an RBAC token negotiation handled by the client library; once connected, commands are sent as plain ASCII strings.

Like SCRP, NCRP defines only set and get commands, but extends them with a DEVICE field that identifies the target device within the gateway. A blank DEVICE field addresses the gateway's own built-in device (physical address zero); otherwise the device is specified

as a symbolic name of up to 23 characters or as a 32-bit numeric address. This addressing scheme makes it possible to issue commands to multiple devices on the same gateway over a single TCP connection.

NCRP includes a TAG mechanism to support pipelined operation. The TAG is an opaque client-assigned string of up to 31 characters, prepended to each command and echoed verbatim at the start of every response. Because the server may return responses out of order relative to the commands that triggered them, the client uses the TAG to correlate each response to its originating request. A successful set response has the form `$TAG .\n\n;`, a successful get response appends the property data before the terminal delimiter, and an error response carries an error number and message in place of the data.

Despite this protocol-level support for concurrent in-flight commands, the `pyfgc 1.x` and `2.x` implementations used NCRP strictly sequentially: each command was issued individually, the caller blocked until its response arrived, and the TAG mechanism was not exposed in the public API. NCRP has therefore served as the primary protocol for FGC control software, but in practice was operated in a synchronous, one-command-at-a-time mode.

Both SCRP and NCRP use the FEC Command String format [22] to identify the target property and qualify the request. A command string consists of a mandatory property name followed by up to three optional fields, which may appear in any order: a cycle selector in parentheses (e.g. `(SPS.USER.MD5)`), a transaction identifier in angle brackets (e.g. `<10>`), and a filter in square brackets (e.g. `[0:10]`), used to select a sub-range for array-valued properties. A get command string may end with a list of options separated by whitespace, while a set command string may end with the value to be written. For example, `REF.PULSE.AMPLITUDE(SPS.USER.MD5)<10>` requests the value of `REF.PULSE.AMPLITUDE` for the `SPS.USER.MD5` cycle, tagging the response with transaction ID 10. The filter can be combined with the other fields, as in `REF.PULSE.AMPLITUDE[0:10]<10>`, which restricts the response to the first ten elements of an array-valued property while tagging it with transaction ID 10. A set command follows the same structure but ends with the value to be written, as in `REF.I.DC 100.0`, which sets `REF.I.DC` to `100.0`. As CERN's control infrastructure has evolved, several limitations of NCRP have become apparent beyond the unused pipelining capability. It does not natively support subscription-based notifications, session initialization handshakes, or remote terminal access. Furthermore, being based on raw TCP, it is not directly accessible from modern web browsers or lightweight scripting environments. For these reasons, NCRP is being superseded by the FGC Front-End Protocol (FFP) for FGC4, which addresses all of the above shortcomings [24].

2.4. Authentication and Authorization: RBAC

Role-Based Access Control (RBAC) provides the authentication and authorization infrastructure for the CERN accelerator control system [12, 13]. It is used by all major device access protocols at CERN, including RDA, NCRP, and FFP, to verify the identity of client applications and to enforce per-device, per-property access policies. The system is divided into two independent components: *Authentication* (A1) and *Authorization* (A2), which interact only through the signed token that A1 issues to the client and the client subsequently presents to A2.

The A1 subsystem verifies the digital identity of a principal, either a human operator or an automated program. The A1 server is implemented in Java, receives authentication requests via HTTP, and returns a digitally signed RBAC token on success. Every request is stateless: the server keeps no session and each token contains the full set of information needed for downstream authorization without further contact with A1. The token fields are listed in table 2.3.

Table 2.3: Fields of an RBAC authentication token [12].

Field	Type	Description
ID	int	Token identification number
User	string	CERN account name
Roles	string[]	Active roles assigned to the user
AuthTime	int	Unix timestamp of authentication
ExpTime	int	Unix timestamp of token expiry
Application	string	Name of the requesting application
Critical	boolean	Whether the application is flagged as critical
Location	string	Logical name of the user's network location
Address	bytes	IP address of the user's host
Signature	bytes	SHA1+RSA 512-bit signature over the token body

Tokens are short-lived: application tokens expire after at most 720 minutes (12 hours). A token may carry a *renewable* flag, which allows the RBAC server to issue a refreshed token without requiring the user to re-enter credentials, up to a maximum renewal window of one year. Separately, a *master token* stored on the local machine can be used to derive shorter-lived application tokens on demand, supporting single sign-on for operator workstations.

The A1 server supports five authentication methods [12]. *Location* authentication is the most common for interactive use on the CERN technical network: the server detects that the client's host belongs to a trusted RBAC location and issues a token automatically,

requiring only the CERN account name. *Explicit* authentication uses a username and password submitted via HTTP. *Kerberos* authentication exchanges a Kerberos 5 service ticket for the RBAC service principal. *SAML* authentication uses a SAML response from a federated identity provider. *OAuth 2.0* authentication exchanges an OIDC access token issued by CERN's identity infrastructure.

Authorization The A2 subsystem determines whether a given operation is permitted on a given device property. It is implemented as a C++ library embedded in each device server (FESA, FGC gateway, and others) and performs all checks locally without contacting any external service. At startup the device server loads an *access map*: a tab-separated text file on NFS, digitally signed to prevent tampering, that lists all access rules for that server. Access maps can be reloaded at runtime via the CMW Admin GUI or the `rbac-maps-fec/rbac-maps-class` command-line tools without restarting the server [13].

Access rules are defined per device class in CCDE (Controls Configuration Data Editor) by equipment specialists. Each rule binds a set of device groups and property groups to a required role, a required network location, and optionally a requirement for a critical-flagged token. The A2 library represents the compiled rules as a tree, allowing the authorization check for any operation to be performed in $O(\log n)$ time with respect to the number of rules [12].

Three checking policies govern how strictly the device server applies these rules:

- **NO-CHECK**: no authorization check is performed and no token is required. Used in development and test-bench environments.
- **LENIENT**: the default policy for non-LHC (Injector) front-end computers. Authorization is checked only for device/property pairs that have an explicit access rule; unprotected properties are accessible without a token.
- **STRICT**: the default policy for LHC front-end computers. A valid RBAC token is required for all access, including read-only operations; all Set operations are checked against the access map regardless of whether the property carries an individual rule.

RBAC in the FGC control stack In the NCRP-based FGC2 and FGC3 stack, the RBAC token is transmitted as part of the TCP connection handshake before any commands are issued. The client library serialises the token into NCRP's binary framing and the FGC gateway verifies the signature and applies the configured checking policy before accepting commands.

2.5. FESA

The Front-End Software Architecture (FESA) is a C++ framework developed at CERN for writing device servers that expose accelerator equipment over the CMW/RDA layer [8]. A FESA device server runs on a front-end computer in the TN and publishes the state of one or more physical devices as named properties conforming to the Accelerator Device Model. Client applications on the GPN interact with the equipment exclusively through RDA calls directed at the FESA server; the server translates these into hardware-level commands and returns the results. FESA handles the lifecycle of the server process, the threading model, the RDA registration, and the RBAC access control hooks, leaving the equipment-specific logic to the device class developer.

2.6. Pre-existing software for FGCs

2.6.1. Firmware

FGC devices are governed by firmware implementing the real-time current control loop, including reference generation, feedback regulation, fault management, and device state handling. For FGC4, the firmware runs on the FGCDv2 (FGC Daemon v2) framework [30], a common software platform providing the runtime environment for all software components deployed on FGC4 devices, including the FFP server.

2.6.2. Control Software

A set of software tools is maintained at CERN to interact with FGC devices, ranging from low-level programmatic interfaces to operator-facing applications. These tools are accessible through the FGC Portal, a web-based launch pad that provides a single entry point to all applications in the ecosystem, as shown in fig. 2.9.

FGC API The FGC API is a language-agnostic REST web service (version 1.10.0, OpenAPI 3.1) that provides a centralized and documented interface for accessing FGC-related data and sending commands to devices. Its endpoints expose entity topology (name files and group files), property definitions, device class information, and FGC database records. By decoupling the API surface from the underlying data sources, applications can retrieve FGC configuration and interact with devices without being bound to a specific backend implementation.

A significant advantage of this architecture is that the FGC API resolves a networking

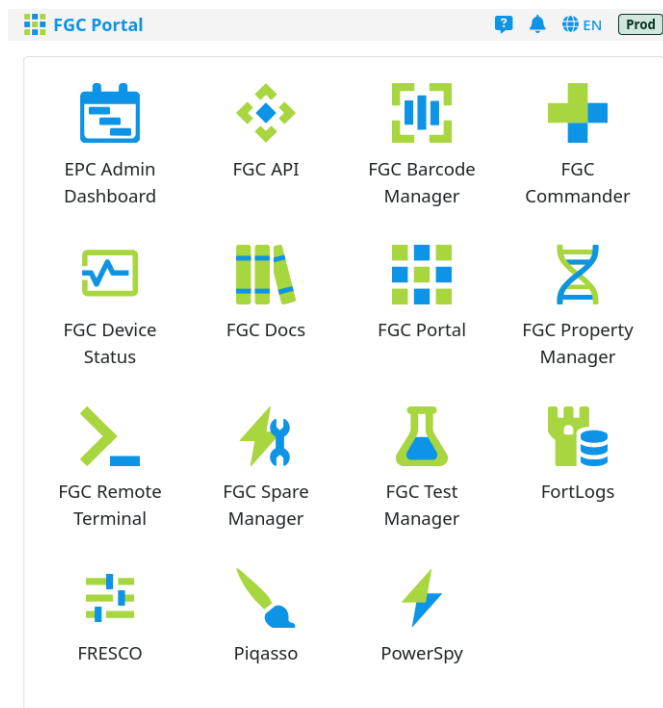


Figure 2.9: The FGC Portal, providing a unified entry point to all applications in the FGC control software ecosystem. The selector in the top right corner allows developers to switch between deployment environments (dev, qua, prod).

constraint present in direct device communication: whereas `pyfgc` requires access to the CERN Technical Network to reach FGC gateways, the FGC API is exposed as a standard HTTPS service and is therefore accessible from any network, provided the caller holds a valid CERN account. Authentication and authorization are enforced at the API boundary via RBAC [12], so all access-control guarantees are preserved without requiring clients to manage tokens or maintain a direct connection to the control network. The FGC API serves as the common device interaction and data layer for the higher-level applications available through the FGC Portal, including FGC Commander and FRESCO.

FGCRun+ Before FGC Commander, the primary tool for monitoring and commanding FGC devices was FGCRun+, a desktop application written in Perl using the Tk graphical toolkit. Running directly on dedicated CERN workstations and communicating with FGC gateways over NCRP, FGCRun+ provided device monitoring, command dispatch, subscription-based real-time updates, and diagnostics for the entire FGC fleet.

FGC Commander FGC Commander is the web application that replaced FGCRun+. It is used by engineers and operators to monitor and command FGC devices from any browser without requiring local installation. It provides a hierarchical device browser with regular-expression search, allowing users to select devices across all accelerator areas and device generations. The main view displays real-time device state indicators (operational state, voltage/current status, polarity) alongside live measurements of current reference, current measurement, voltage reference, and voltage measurement for all selected devices, as illustrated in fig. 2.11. FGC Commander delegates all device interaction to the FGC API, which in turn uses `pyfgc` to communicate with the underlying FGC devices. Real-time device state indicators are obtained via the FGC status server through `pyfgc_statussrv`. The FGC Commander backend is responsible only for user-specific metadata such as device tags, device class configuration, and permalinks.

PowerSpy PowerSpy is a web application for visualizing real-time and offline data from CERN power converters and SVCs. It handles data in the form of analog and digital time series with periods down to one microsecond, as well as text data in tabular form. The system is designed to scale to order of 100 concurrent users, order of 10^7 data points per session, and order of 10^3 data sources. PowerSpy was initiated in 2014 as a replacement for the existing LabView Data Viewer, and has evolved through five development phases. The current version features a Vue.js frontend, FFT spectrum analysis (fig. 2.12), integration with the FortLog acquisition database, and support for nanosecond-resolution timestamps.

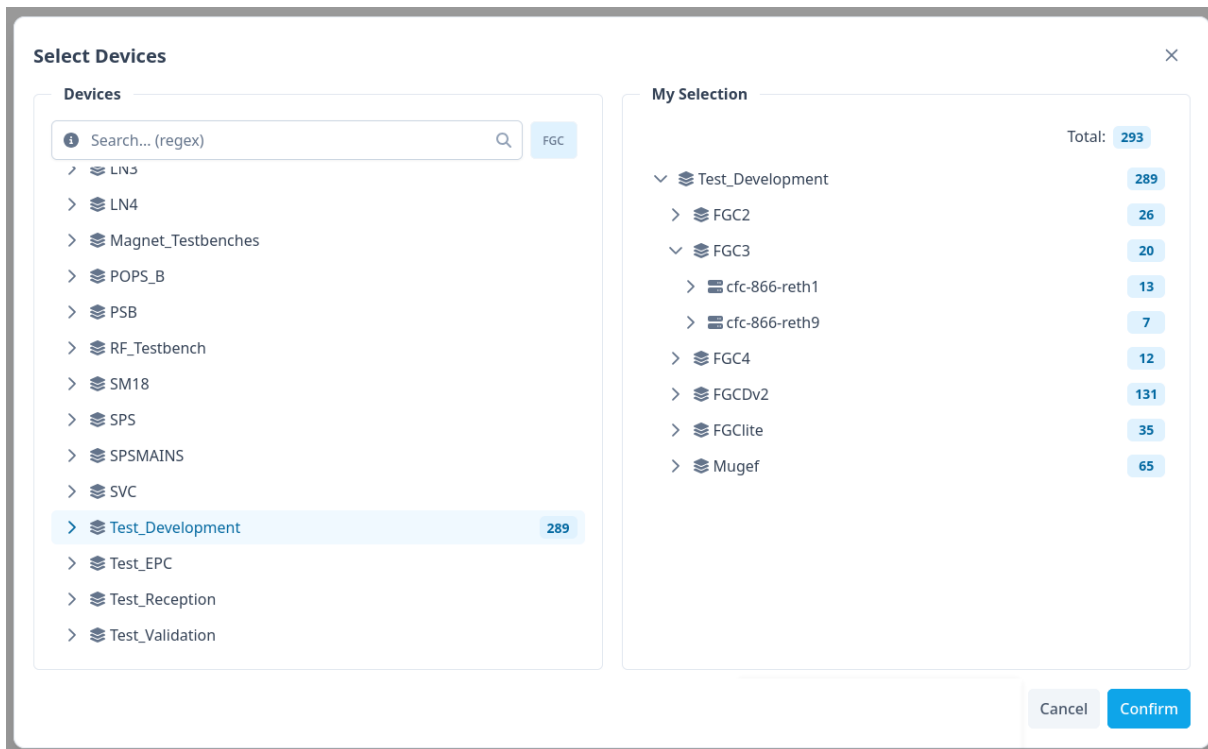


Figure 2.10: FGC Commander device selection dialogue. The left panel lists all available device groups with regular-expression search; the right panel shows the current selection broken down by device type (in this case, FGC generation).

FGC Commander					Connect	Disconnect	Select	Gateway	FGC	Config	Database	Analysis	Help
PL	OP	VS	PC	I_REF	I_MEAS	V_REF	V_MEAS	FGC	6 / 6	...	>	🕒	
LK	NL	RD	IL	661.59	661.59	502.63	510.23	⚡	RPADO.SR1.RD1.LR1				
LK	NL	FS	FO	0.00	0.00	0.00	-0.01	⚡	RPADO.SR1.SPARE.1				
LK	NL	RD	DT	0.00	0.00	18.00	0.00	⚡	RSVCF.SEQ2.SVC.LHC2				
LK	NL	RD	DT	0.00	0.00	18.00	0.00	⚡	RSVCG.SEQ6.SVC.LHC6				
LK	NL	RD	DT	0.00	0.00	18.00	0.00	⚡	RSVCI.SEQ4.SVC.LHC4				
LK	NL	RD	DT	0.00	0.00	18.00	0.00	⚡	RSVCJ.SEQ8.SVC.LHC8				

Figure 2.11: FGC Commander monitoring view, showing real-time device state and measurements for a set of selected FGC devices. The colour-coded status indicators (operational state, voltage/current status, polarity) are obtained via the FGC status server, while the current and voltage reference and measurement values are read directly from the devices.

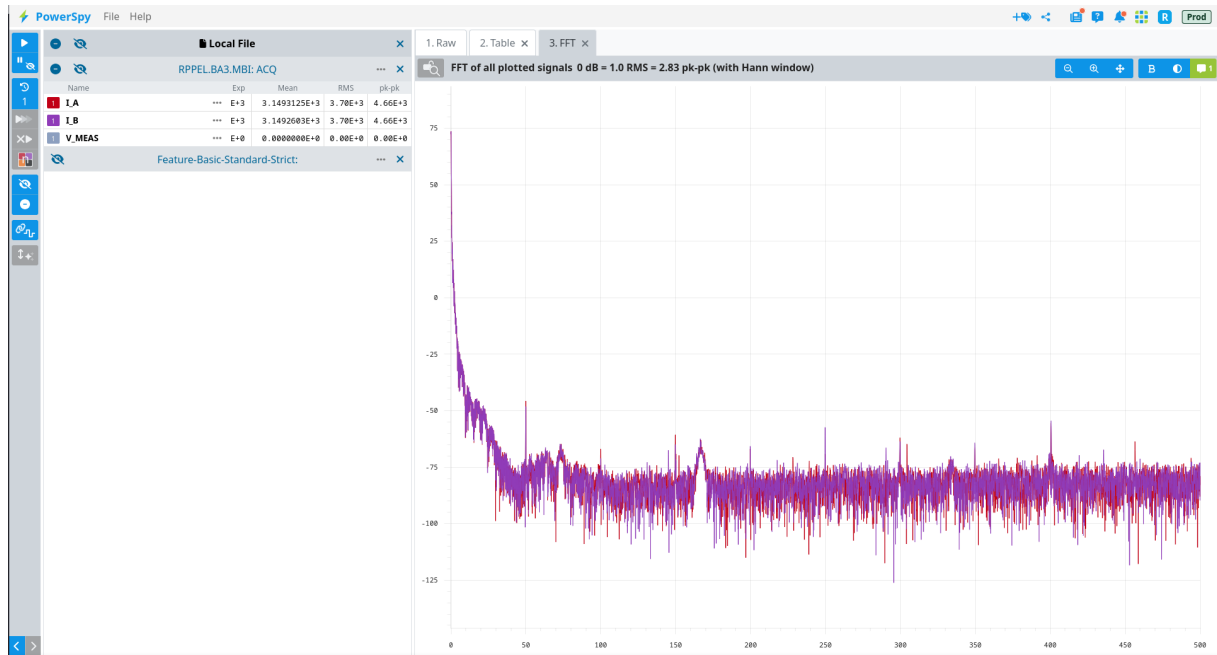


Figure 2.12: PowerSpy displaying an FFT analysis of current and voltage signals acquired from a power converter. The application supports raw time series, tabular, and frequency-domain views.

FRESCO FRESCO (version 1.0.6) is a web application for automated experimentation on FGC devices. Users define *recipes*, which are parameterized sequences of operations executed on one or more FGC devices: setting a target regulation mode (current or voltage), ramping the current reference, applying known perturbations, and acquiring the resulting signals. FRESCO runs each recipe in a dedicated subprocess, manages RBAC token renewal across long-running experiments, and stores results for later analysis. It was created by Benjamin Raymond during PowerSpy Phase IV (2020–2022) and subsequently developed further by Vasile Sambor and Stavroula Livarda.

2.6.3. ACC-Py

Accelerator Python (ACC-Py) is CERN’s Python infrastructure service for the accelerator controls domain [6]. It provides a curated Python base distribution pinned to a tested set of third-party packages, an internal package index (analogous to PyPI but accessible only within the CERN network), GitLab CI job templates for building, testing, and publishing Python packages, and a documentation hosting service. All Python packages in the FGC ecosystem are distributed through the ACC-Py package index and their CI pipelines are built on the ACC-Py GitLab CI template library ([acc-co/devops/python/acc-py-gitlab-ci-templates](https://github.com/acc-co/devops/python/acc-py-gitlab-ci-templates) [7]).

2.6.4. Python client libraries

Communication between control applications and FGC devices, as well as the supporting services described above, is implemented through a small ecosystem of Python packages, of which `pyfgc` is the central component. It is responsible of opening the connection to a device and carrying out the requests, relying on `pyfgc_name` to resolve the device name to a gateway address and on `pyfgc_rbac` to obtain the RBAC token presented during the connection handshake. `pyfgc_statussrv` and `pyfgc_decoders` serve a related but separate purpose: rather than issuing commands to a device, they consume the periodic status broadcasts from an FGCD gateway and decode them into per-device state information, and are used for this purpose by the FGC API and FGC Commander. `pyfgc`, `pyfgc_name`, and `pyfgc_rbac` are the primary subject of the rewrite described in this thesis.

pyfgc The `pyfgc` Python library (versions 1.x and 2.x) is the pre-existing Python client for communicating with FGC devices over NCRP or SCRP. It exposes a procedural API of module-level functions (`connect()`, `get()`, `set()`, `fgc()`) that open connections, issue commands, and return responses.

pyfgc_name Before a connection can be opened, the gateway address corresponding to a device name must be determined. `pyfgc_name` (version 2.1.1) performs this resolution by parsing two configuration files: the name file, which maps each device to its gateway address and class identifier, and the group file, which assigns gateways to named accelerator groups and timing domains. These files are maintained in a dedicated GitLab repository [27].

pyfgc_rbac Once the gateway is known, `pyfgc` must authenticate the connection. `pyfgc_rbac` (version 1.7.0) is the Python client for the CERN RBAC authentication service, providing functions to acquire RBAC tokens via all supported authentication methods (location, Kerberos, OAuth 2.0, SAML, and username/password), as well as utilities to renew tokens and reduce their role set. Tokens are returned as opaque byte strings and passed directly to `pyfgc` during the NCRP connection handshake.

pyfgc_statussrv Separately from the request/response path, `pyfgc_statussrv` (version 1.0.0) is a client for the FGC status server, a gateway-level service that aggregates and broadcasts the real-time operational status of all devices connected to an FGCD gateway. It is consumed by both the FGC API and FGC Commander to populate device state indicators, and exposes two functions: `get_status_device()`, which retrieves and decodes the status record for a single named device, and `get_status_all()`, which retrieves the

full status block for all devices on the gateway. Decoded results include per-device fields such as operational state, fault flags, and timestamps. The binary decoding is delegated to `pyfgc_decoders`.

pyfgc_decoders `pyfgc_decoders` is a utility package containing binary decoders for the data types used in FGC status and property responses. It provides both a common decoder, applied to all device classes, and class-specific decoders that handle the proprietary encoding of each FGC firmware class. The package predates this work and is used as a dependency by `pyfgc_statussrv`.

The ecosystem described in this chapter provides the foundation for FGC3 communication but leaves FGC4 entirely unsupported. The following chapter analyses the specific functional gaps and structural deficiencies that must be addressed to extend this ecosystem to FGC4.

3 | Problems

3.1. Problems with current solutions

The `pyfgc` ecosystem that predated this work exhibited limitations at two distinct levels: *functional gaps*, which made it impossible to communicate with FGC4 devices, and *software quality deficiencies*, which made the codebase difficult to maintain, test, and extend. The quality deficiencies share a common origin: the original FGC control software was written in Perl, and the Python packages were produced by migrating that code while preserving its programming style. Perl idioms such as procedural function sets, module-level global variables, and the absence of encapsulating objects were carried directly into Python, producing code that is syntactically Python but architecturally Perl. The consequences of this stylistic inheritance run through several of the problems described below. Both categories are addressed by the rewrite described in chapter 4.

3.1.1. Lack of FGC4 support

The most fundamental problem is that the existing `pyfgc` library provides no mechanism to communicate with FGC4 devices. The library supports two communication channels: a synchronous TCP/IP channel (used with NCRP) and a serial channel (used with SCRP). WebSockets, the transport required by the FFP protocol, are not implemented. Additionally, no FFP protocol class exists in the library: the only application protocols available are NCRP, SCRP, and RDA.

The consequence is total: any attempt to connect to an FGC4 gateway fails at the channel level, before any protocol negotiation takes place. Integrating FGC4 into an existing `pyfgc`-based control application would require rewriting the transport and protocol layers from scratch, which is precisely what this work does.

3.1.2. Implicitly synchronous communication

The `pyfgc` 1.x and 2.x API is built around a strictly sequential request/response model. Each call to `pyfgc.get()` or `pyfgc.set()` opens a connection, transmits one command, blocks the calling thread until the single response arrives, and returns. In `pyfgc` 1.x the convenience top-level functions even create and destroy a session on each call:

```
1 def get(target: str, prop: str, get_option: str = None,
2         protocol: str = "sync", **kwargs):
3     # A new FgcSession is created and destroyed on every call.
4     with fgc(target, protocol=protocol, **kwargs) as fgc_session:
5         return fgc_session.get(prop, get_option)
```

Listing 3.1: Top-level `get` function in `pyfgc` 1.x. A full session is created and torn down for every property access (source/old/pyfgc-1.8.6/pyfgc/api.py).

This design prevents any form of concurrent operation. A control application that needs to poll several devices, or that must issue multiple commands to the same device without waiting for each acknowledgement, cannot do so with this API. Although the NCRP protocol itself includes a TAG mechanism that allows multiple commands to be in flight concurrently and responses to arrive out of order (see section 2.3.3), `pyfgc` 1.x and 2.x never leveraged this capability: the TAG was not exposed in the API and no pipelining was ever performed. FFP is designed to be fully asynchronous by default: any number of requests may be in flight simultaneously, with responses arriving in any order. The existing synchronous model is architecturally incompatible with this requirement.

3.1.3. Global mutable state

The packages in the old ecosystem rely heavily on module-level global variables. In `pyfgc_rbac` 1.x, the RBAC server address, port, and cryptographic parameters are stored as module globals, modified via setter functions:

```
1  # Global constants -- shared across the entire process
2  SERVER = "rbac-pro-lb.cern.ch"
3  PORT   = 8443
4
5  def set_rbac_server(rbac_server):
6      global SERVER
7      SERVER = rbac_server
8
9  def set_rbac_port(rbac_port):
10     global PORT
11     PORT = rbac_port
```

Listing 3.2: Module-level globals in `pyfgc_rbac 1.x` (`source/old/pyfgc_rbac-1.7.0/pyfgc_rbac/pyfgc_rbac.py`).

In `pyfgc_name 2.x`, the parsed namefiles are equally stored in module-level dictionaries (`gateways`, `devices`, `groups`), each guarded by the same `try/except NameError` pattern:

```
1  try:
2      gateways
3  except NameError:
4      gateways = dict()
5
6  try:
7      devices
8  except NameError:
9      devices = dict()
10
11 try:
12     groups
13 except NameError:
14     groups = dict()
```

Listing 3.3: Module-level dictionaries in `pyfgc_name 2.x` (`source/old/pyfgc_name-2.1.1/pyfgc_name/pyfgc_name.py`). Five separate globals hold the entire device database.

A `reset()` function clears all of them. This design has two concrete consequences. First, it is impossible to maintain two independent name resolutions in the same Python process, for instance when a caller needs to load namefiles for two different accelerator sectors or to combine production data with a custom set used for testing. Second, unit tests are not isolated: any test that modifies the global state contaminates the state seen by subsequent tests.

3.1.4. Procedural API and implicit shared state

The public interface of `pyfgc 1.x` is almost entirely procedural: the library exposes a set of module-level functions (`get()`, `set()`, `connect()`, `fgc()`) with no encapsulating object that a caller instantiates or configures. Because there is no object, any state that must persist across calls is necessarily stored at module level. The channel pool is the clearest example: `pyfgc` maintains a single process-wide dictionary of open connections, manipulated through a pair of free functions:

```

1  # Module-level pool shared across the entire process
2  try:
3      _channels
4  except NameError:
5      _channels = dict()
6
7  _lock = threading.Lock()
8
9  def get_channel(fgc, channel_type, gateway, rbac_name, **kwargs):
10     with _lock:
11         if (rbac_name, fgc, channel_type, gateway) in _channels:
12             _channels[rbac_name, fgc, channel_type, gateway]['count']
13                 += 1
14         else:
15             _channels[rbac_name, fgc, channel_type, gateway] = {
16                 'count': 1,
17                 'channel': ChannelFactory.create(...)
18             }
19     return _channels[rbac_name, fgc, channel_type, gateway]['channel']

```

Listing 3.4: Module-level channel pool in `pyfgc 1.x` (source/old/pyfgc-1.8.6/pyfgc/channel_manager.py). All callers in the process share the same `_channels` dictionary.

The `try/except NameError` guard is itself a symptom of the design: it exists to prevent the pool from being reset if the module is re-executed, a fragility that would not arise with a class instance. Because `_channels` is a module-level singleton, it is impossible for two independent parts of the same process to maintain separate connection pools, for instance one connecting to the operational network and one to a test bench. Equally, a unit test that exercises `get_channel()` leaves state in `_channels` that is visible to every subsequent test in the process. Replacing the module-level pool with a singleton class, each instance owning its own data structure, eliminates both constraints without any change to the

underlying connection logic. This pattern, where the absence of encapsulating objects forces state into module globals, is part of a wider OOP gap in the ecosystem: the same design choice that produces implicit shared state also prevents the library from offering a structured `RBACToken` type (see section 3.1.5) or a unified high-level connection API (see section 3.1.11).

3.1.5. Absence of a structured RBAC token type

The `pyfgc_rbac` 1.x package provides no structured representation of an RBAC token. Every authentication function (`get_token_location()`, `get_token_kerberos()`, `get_token_oauth2()`) returns an opaque byte string. The package offers no method to inspect the token's validity period, check whether it has expired, or compute the time remaining before it does. Any caller that needs this information must decode the binary payload manually by calling `pyfgc_rbac.token_to_dict(token)` and extracting the `ExpirationTime` field. The dictionary returned by `token_to_dict()` is completely untyped and undocumented: its keys are plain strings whose names and value types are not specified anywhere in the `pyfgc_rbac` API, leaving callers to infer the schema from the token payload itself or from internal library code. This decoding logic is not part of `pyfgc_rbac`: it was duplicated directly into `FgcSession` in `pyfgc` 1.x, as shown in listing 3.5.

```

1  # pyfgc_rbac 1.x: returns bytes -- no expiry information exposed
2  def get_token_kerberos():
3      ...
4      return _request_token(form) # returns bytes; caller cannot check
                                   validity
5
6  # pyfgc 1.x: every consumer must decode the token payload manually
7  def _get_token_expiration_time(token):
8      if not token:
9          return None
10     return datetime.fromtimestamp(
11         int(pyfgc_rbac.token_to_dict(token)['ExpirationTime']))

```

Listing 3.5: `pyfgc_rbac` 1.x returns tokens as opaque bytes (top); `pyfgc` 1.x must decode the payload itself to determine expiry (`source/old/pyfgc-1.8.6/pyfgc/fgc_session.py`) (bottom).

Any service that uses `pyfgc_rbac` independently (such as FGC API) must either replicate this decoding logic or accept that tokens will eventually expire silently. The tight coupling between the token byte format and the consumers that parse it creates a fragile contract: if the token encoding changes, every caller that manually decodes it must be updated

separately. This problem is an instance of the broader weak-typing pattern discussed in section 3.1.7: the token is a domain object with a well-defined structure, yet the library represents it as opaque bytes because the procedural design of section 3.1.4 provides no natural place for the associated behaviour.

3.1.6. Code duplication between synchronous and asynchronous paths

`pyfgc 2.x` introduced a distinction between a `SyncProtocol` and an `AsyncProtocol` class hierarchy, aiming to support both blocking and non-blocking usage. In practice, the result is near-identical `get()` and `set()` implementations in both hierarchies:

```

1  # SyncProtocol (source/old/pyfgc-2.0.3/pyfgc/protocols/sync_protocol.
    py)
2  def get(self, property_name: str,
3          get_option: str = None) -> AsyncSyncResponse:
4      encoded_command = AsyncSyncCommandEncoder.encode_get(
5          self.target_fgc, property_name, get_option)
6      sent_command_time = time.time()
7      try:
8          raw_response = self._send_command_to_target(encoded_command)
9      except BrokenPipeError as e:
10         super().destroy_channel()
11         raise ChannelError(e)
12     return AsyncSyncResponse(raw_response, self.target_fgc,
13                             property_name,
14                             timestamp_sent=sent_command_time)
15
16 # AsyncProtocol (source/old/pyfgc-2.0.3/pyfgc/protocols/
    async_protocol.py)
17 async def get(self, property_name: str,
18               get_option: str = None) -> AsyncSyncResponse:
19     if not self.channel:
20         raise ChannelError("Channel is invalid!")
21     sent_command_time = time.time()
22     raw_response = await self.channel.get(
23         property_name, get_option, self.target_fgc)
24     return AsyncSyncResponse(raw_response, self.target_fgc,
25                             property_name,
26                             timestamp_sent=sent_command_time)

```

Listing 3.6: Duplicated `get` implementations in `SyncProtocol` and `AsyncProtocol` (pyfgc 2.x). The logic is identical; only the `await` keyword and the internal channel call differ.

Any bug fix or protocol change must be applied separately to each hierarchy. As the number of supported protocols and transport layers grows, and their functionality is expanded, this pattern makes the codebase increasingly difficult to maintain.

3.1.7. Weak type safety and string-based dispatch

Protocol and channel selection in pyfgc 2.x is performed by passing string literals to factory functions. The `FgcSession` constructor accepts a `protocol` argument of type `str` and resolves it against a module-level dictionary:


```
1  PROTOCOL_CLASSES = {  
2      ProtocolTypes.ASYNC.value: AsyncProtocol,  
3      ProtocolTypes.SERIAL.value: SerialProtocol,  
4      ProtocolTypes.SYNC.value: SyncProtocol,  
5  }  
6  # ...  
7  # Usage: FgcSession("DEVICE", "sync")  
8  # Passing an unsupported string fails silently until runtime.
```

Listing 3.7: String-based protocol dispatch in `pyfgc 2.x` (source/old/pyfgc-2.0.3/pyfgc/fgc_session.py).

There is no static guarantee that the string passed by the caller names a valid protocol, nor that the chosen protocol is compatible with the chosen channel. Adding a new protocol requires modifying the dispatch dictionary in addition to writing the new class, coupling the two concerns.

The same absence of typed domain objects recurs throughout the ecosystem. In `pyfgc-rbac 1.x`, every authentication function returns a raw `bytes` value: there is no named type, no documented structure, and no way for a type checker or an IDE to tell a caller what the bytes contain or whether the token is still valid (see section 3.1.5). In `pyfgc_name 2.x`, the device database is exposed as plain dictionaries: `gateways` and `devices` map string keys to further `dict` values, so callers must know the key names by convention, and any access to a missing key fails only at runtime with a `KeyError`. The nesting goes several levels deep: each entry in `devices` is itself a plain `dict` built with successive key assignments, and the `sub_devices` field it may contain is a list of further plain dicts. The `groups` dictionary is a recursively nested tree of plain dicts, assembled by walking a path of group names and creating an empty dict at each missing level. No type annotation, no dataclass, and no schema document describes what keys any of these dicts are expected to contain:

```

1  # Every device entry is an ad-hoc dict; keys are string literals
   scattered
2  # across the parsing function -- no schema or type defines the
   structure.
3  cur_devices[devname] = dict()
4  cur_devices[devname]["name"]      = devname
5  cur_devices[devname]["channel"]   = channel
6  cur_devices[devname]["class_id"]  = class_id
7  cur_devices[devname]["omodemask"] = omodemask
8  cur_devices[devname]["gateway"]   = hostname # added later in the
   same loop
9
10 # Sub-devices are a list of further plain dicts, with None as a
   sentinel:
11 sub_device = dict()
12 sub_device["alias"] = alias
13 sub_device["name"]  = device + ":" + str(channel)
14 cur_devices[device]["sub_devices"][channel] = sub_device
15
16 # Groups form a recursively nested plain-dict tree; no fixed depth or
   schema:
17 for g in groups_list:
18     if g not in groups_temp:
19         groups_temp[g] = dict()
20     groups_temp = groups_temp[g]
21 groups_temp["gateways"].append(gw_host)

```

Listing 3.8: Nested plain-dictionary construction in pyfgc_name 2.x (source/old/pyfgc_name-2.1.1/pyfgc_name/pyfgc_name.py). The shape of each entry is implicit in the parse logic; there is no named type.

Both are consequences of the procedural design identified in section 3.1.4: when state lives in module-level variables rather than in class instances, the type annotations and encapsulation that classes provide are unavailable, and data naturally ends up represented as generic containers instead of domain-specific objects.

3.1.8. Legacy Python version and absence of code quality tooling

The old packages target Python 3.6, a version released in 2016 and no longer maintained since 2022. Modern language features that improve correctness and readability are absent: there are no `dataclasses`, no `TypedDict`, no `X | Y` union type syntax, no `Self` type annotation, and no structural pattern matching. Type hints, where present at all, are sparse and inconsistent:

```
1 def get_token_location():
2     try:
3         response = _request_token()
4         return response
5     except:
6         raise RbacServerError("Could not get location token")
```

Listing 3.9: A representative function from `pyfgc_rbac` 1.x with no type annotations and a bare `except` clause.

None of the packages apply a code formatter (Black, isort) or a linter, leading to inconsistent style across files. Differences in indentation, import ordering, and naming conventions add friction when reading or reviewing the code.

3.1.9. Absence of docstrings and documentation

Most functions and classes in `pyfgc` 1.x and `pyfgc_rbac` 1.x carry no docstrings. There is no generated API documentation and no user-facing guide or tutorial. A developer who wants to understand the expected arguments, return type, or error conditions of a function must read its implementation. In a library intended for use across multiple teams at CERN, this is a significant usability barrier.

Where documentation does exist, it is outdated or incomplete to the point of being unusable. The README of `pyfgc` 1.x and 2.x, unchanged between the two versions, reads in full:

pyfgc is a Python library to interface Function Generation Controllers (FGCs) via TCP/IP, via a front-end computer; and/or USB ports, plugged directly into your computer.

This description does not mention the protocol and channel abstractions introduced in `pyfgc` 2.x, refers to USB ports where the library in fact implements a serial channel, and provides no information on installation, configuration, or usage. The `pyfgc_name` 2.1.1 README is empty. The `pyfgc_statussrv` 1.0.0 README lists a table of contents with the entries “Main features”, “Installation”, and “Usage”, but no content follows any of these headings. No code example anywhere in the ecosystem demonstrates how to perform a basic operation such as connecting to a device and reading a property value.

3.1.10. Tightly coupled code

The `FgcSession` class in `pyfgc 2.x` conflates several distinct responsibilities in a single constructor: device name resolution (calling `pyfgc_name`), RBAC token acquisition, protocol selection via string dispatch, channel instantiation, and initial connection setup. None of these concerns can be tested in isolation without also exercising all the others. Adding a new concern, such as support for a new protocol, requires modifying the same monolithic constructor and the classes it instantiates.

3.1.11. No high-level connection API

The old `pyfgc` ecosystem provides no convenience entry point that hides the details of name resolution, protocol selection, and token acquisition from the caller. Every application that opens a connection to an FGC device must perform three manual steps: resolve the device name to a gateway address by calling `pyfgc_name`, acquire an RBAC token using `pyfgc_rbac`, and then pass the name file path, the token bytes, and a protocol string to the `pyfgc` session:

```

1  # pyfgc 1.x: three concerns orchestrated manually by the caller
2  import pyfgc_rbac, pyfgc
3
4  token = pyfgc_rbac.get_token_kerberos()    # token acquired separately
5  with pyfgc.fgc("RFNA.866.01.ETH1",
6                protocol="sync",            # must
7                know protocol
8                name_file="/user/fgc/settings/fgc.names",    # must
9                know file path
10               rbac_token=token) as session:
11     value = session.get("I_MEAS")
12
13 # pyfgc 3.x: protocol selection and name resolution are automatic
14 from pyfgc.autoconnection.factory import AsyncAutoConnectionFactory
15 from pyfgc_rbac import RBACManager
16
17 token = RBACManager().get_kerberos()    # typed RBACToken, not raw
18     bytes
19 conn = await AsyncAutoConnectionFactory.auto_create_connection(
20     "RFNA.866.01.ETH1", rbac_token=token)
21 async with conn:
22     response = await conn.get("I_MEAS")

```

Listing 3.10: Connecting to an FGC3 device in `pyfgc 1.x` requires the caller to manually orchestrate name resolution, token acquisition, and protocol selection (top). `pyfgc 3.x` reduces this to a single call (bottom).

The caller must know the correct `protocol` string for the target device generation (a detail that is not derivable from the device name alone without reading the name file) and the path or URL of the name file. Neither is documented in the API. Adding an FGC4 device to the same application would require changing the protocol string and possibly the name file path, meaning the application code encodes device-generation knowledge that should be internal to the library. This is also a direct consequence of the tight coupling described in section 3.1.10: because `FgcSession` conflates connection, name resolution, and protocol selection in one constructor, none of those concerns can be hidden behind a higher-level abstraction without restructuring the class. The `AutoConnection` layer in `pyfgc 3.x` (see section 4.6) resolves this entirely: it reads the device class from the name file and maps it to the correct channel and protocol automatically, so the caller only needs to supply the device name.

3.1.12. Absence of meaningful automated tests

The pre-existing packages contain virtually no automated test coverage. Each package includes a small set of test files, but these do not exercise functional behaviour. The only test guaranteed to run in the GitLab CI pipeline is shown in listing 3.11: a single function that asserts the Boolean literal `True`. Its docstring makes the intent explicit.

```
1  # Just an empty test for the Gitlab CI to succeed the testing
2  # The remaining tests have to be skipped as they can only run in a
   dev machine
3
4  def test_ci():
5      assert True
```

Listing 3.11: The sole CI-passing test in `pyfgc 1.x` and `2.x` (`source/old/pyfgc-1.8.6/pyfgc/tests/test_ci.py`). An identical file exists in `pyfgc 2.x`.

Beyond `test_ci.py`, the `pyfgc 1.x` test suite contains six further files: `test_async.py`, `test_monitor.py`, `test_performance.py`, and `test_serial.py` are entirely commented out, containing no executable code; `test_multi_threading.py` declares three test functions that each contain only a `pass` statement and no assertions; and `test_response.py` is gated on a hostname check that restricts execution to a specific CERN development machine. The remaining tests in `test_sync.py` and `test_api.py` apply the same hostname guard. None of these tests qualify as unit tests: every one of them requires a live FGC device to be reachable on the CERN Technical Network, making automated execution impossible by construction.

The companion packages perform no better: in `pyfgc_rbac 1.x` and `pyfgc_name 2.x`, the only CI-safe test in each package checks that the package version attribute is not `None`: no behaviour is verified. All substantive tests in these packages carry a `skipif` guard that disables them when the environment variable `CI_SERVER` is set, or the same hostname restriction. Several tests in `pyfgc 1.x` are additionally decorated with `@pytest.mark.skip(reason="failing")`, meaning they were written, found to fail, and left disabled rather than fixed.

The practical consequences are significant. Correctness regressions can only be detected through manual integration testing against live devices ("smoke tests"), which requires hardware access and cannot be automated. Any refactoring of internal logic provides no safety net: a change that silently alters behaviour will go undetected until a developer notices unexpected results on a real device, or until the tests are run manually in a development environment. The tight coupling discussed above compounds the problem,

since the absence of seams between components makes it impractical to introduce tests without first restructuring the code. These combined deficiencies make the existing codebase resistant to change and impossible to validate without direct access to live hardware.

The problems identified in this chapter motivate a complete redesign of the `pyfgc` ecosystem. The following chapter presents the architecture and implementation decisions that address each of these deficiencies, and introduces the new components required to support FGC4 communication via FFP.

4 | Content of the thesis

4.1. Idea definition

The problems described in chapter 3 identify a set of concrete requirements for the new `pyfgc` ecosystem. The library must support FFP over WebSockets in order to communicate with FGC4 devices, while retaining backward compatibility with FGC2 and FGC3 via NCRP over TCP/IP. Serial connections via SCRP must be preserved for local direct access. Subscription-based property monitoring via the CMW/RDA stack must also be supported, as operator applications depend on it to observe device property changes without polling. The communication model must be asynchronous by default, allowing multiple requests to be in flight simultaneously; a synchronous interface must also be provided so that users who do not require concurrency can interact with devices through a simple, blocking API without needing knowledge of async programming. Global state must be eliminated, so that connections are explicit objects with well-defined lifetimes. The transport layer and the protocol layer must be decoupled, so that any protocol can be combined with any compatible transport without code duplication. Finally, the codebase must meet contemporary software engineering standards: type annotations throughout, docstrings on all public symbols, comprehensive test coverage, and consistent code style enforced by automated formatters.

These requirements motivate a layered architecture built around two generic abstractions: a *Channel*, which represents the physical transport connection, and a *Protocol*, which handles serialization and deserialization of messages. A *Connection* class composes a Channel and a Protocol into a single object through which the caller sends and receives typed messages. On top of this low-level API sits an *AutoConnection* layer that resolves device names against the namefiles, manages RBAC token lifecycle, and selects the appropriate channel and protocol automatically based on the target device's generation. This design cleanly separates the concerns identified in chapter 3 and makes it straightforward to add new transports or protocols without modifying existing code.

The scope of this chapter extends beyond the Python library. The server-side counterpart

of the FFP stack, the FGCDv2 gateway daemon running on FGC4 hardware, also required additions: the C++ `FfpClient` class was written from scratch, third-party error-handling and integer-type utilities were replaced with standard-library equivalents across the code-base, and the FFP protocol public interface headers were cleaned up. Web-facing control applications built on top of `pyfgc`, namely the FGC API and PowerSpy, were updated to support FGC4 via the new library. DevOps and CI/CD infrastructure was extended in both the Python and web monorepos to support automated testing and containerised deployment. These contributions are described in the sections that follow.

4.2. `pyfgc`: architecture overview

The `pyfgc` 3.0.0 library is organized into four main layers. The *Channels* layer handles the physical transport between the client and the FGC device. The *Protocols* layer handles serialization and deserialization of messages. The *Connection* layer composes a channel and a protocol into a typed send/receive interface. The *AutoConnection* layer provides a high-level, user-facing API that hides all connection details from the caller. Figure 4.1 shows how the packages in the ecosystem interact; fig. 4.2 shows how the four internal layers compose within `pyfgc` itself.

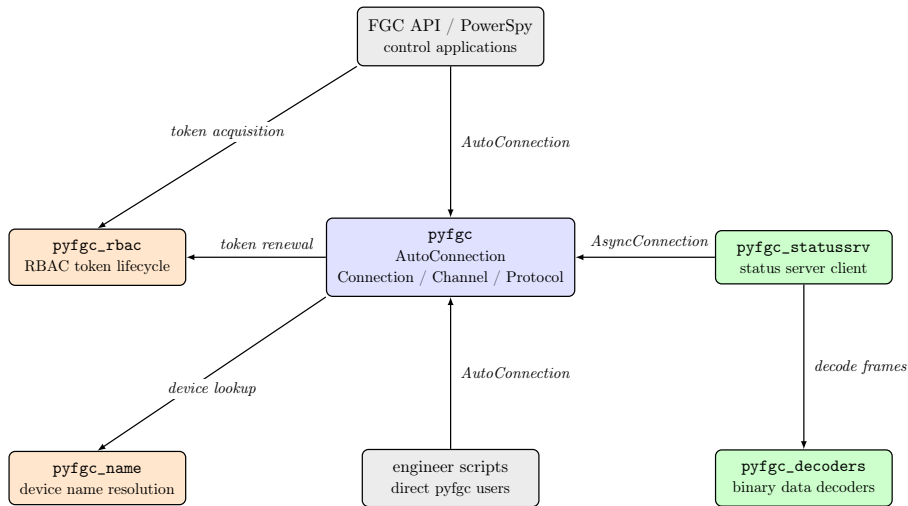


Figure 4.1: Package interactions in the `pyfgc` 3.0.0 ecosystem. `pyfgc` is the central library; its `AutoConnection` layer uses `pyfgc_name` for device resolution and `pyfgc_rbac` for token lifecycle management. `pyfgc_statussrv` uses a `pyfgc AsyncConnection` for network communication and delegates binary decoding to `pyfgc_decoders`. Control applications (FGC API, PowerSpy) and engineer scripts consume `pyfgc` through the `AutoConnection` API and acquire RBAC tokens directly from `pyfgc_rbac`.

Figure 4.2 shows how the four layers compose within `pyfgc`: `AutoConnection` owns a

Connection, which in turn owns a **Channel** and a **Protocol**. Each layer communicates with adjacent layers through a defined interface; the caller interacts only with the top layer.

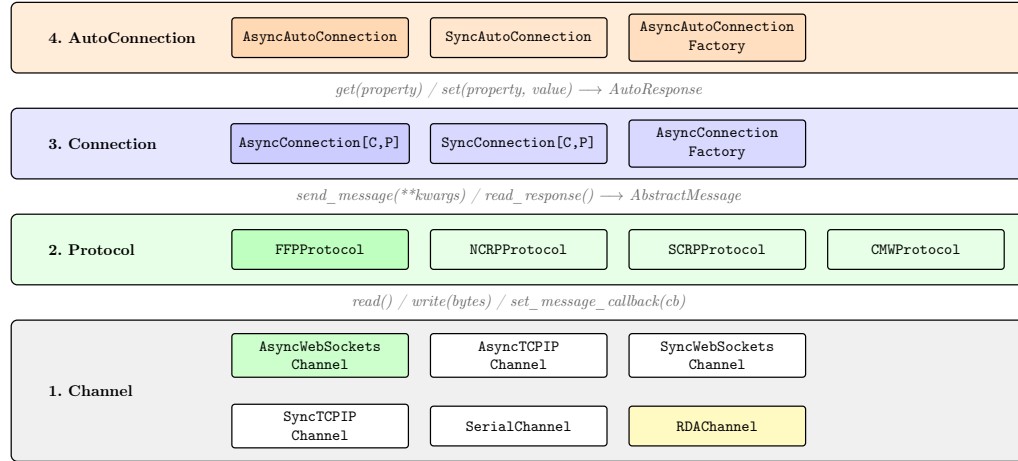


Figure 4.2: Layered component architecture of pyfgc 3.0.0. Each layer communicates with the layer below through a defined interface (shown between rows). **Connection** composes one **Channel** and one **Protocol**; **AutoConnection** owns a **Connection** and adds device name resolution, protocol selection, and RBAC token management. Green fill indicates classes introduced in this version; yellow fill indicates non-standard interface (`RDACChannel`).

All operations are designed as asynchronous coroutines. Synchronous equivalents are generated automatically by wrapping the async code in a `SyncWrapper` utility (described in section 4.8), which runs the coroutine in a dedicated background event loop thread. This design ensures that each synchronous method has exactly one async counterpart, eliminating the code duplication found in pyfgc 2.x. Each of the **Connection** and **AutoConnection** layers exposes both an async and a sync factory class (`AsyncConnectionFactory` / `SyncConnectionFactory` and `AsyncAutoConnectionFactory` / `SyncAutoConnectionFactory` respectively), following the same async-first, sync-wrapped pattern throughout the library.

4.3. Channel layer

The channel layer defines how bytes are exchanged between the Python client and the remote FGC device. The base abstraction is the generic class `AbstractAsyncChannel[M]`, which declares the lifecycle methods (`create`, `destroy`), the I/O primitives (`read`, `write`, `read_until`), and the callback registration interface (`set_message_callback`, `set_disconnect_callback`). The type parameter `M` represents the message type the channel

produces on read, which differs across implementations.

A fully synchronous specialization, `AbstractChannel[M]`, restricts the abstract base so that all methods return concrete values rather than awaitables. Concrete channel classes inherit from either the async base or the sync specialization. Figure 4.3 shows the full class hierarchy.

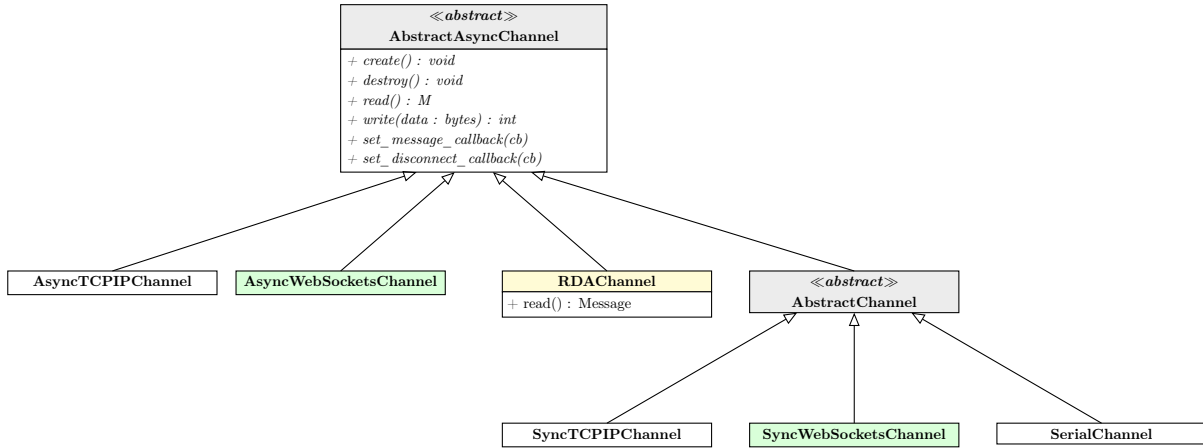


Figure 4.3: Channel class hierarchy in `pyfgc 3.0.0`. `AbstractAsyncChannel` defines the generic transport interface parameterised over message type `M`; `AbstractChannel` is a synchronous specialisation that restricts all return types to concrete values. `RDACHannel` overrides `read()` to return a `Message` object rather than raw bytes. Green fill indicates classes introduced in this version; yellow fill indicates non-standard interface.

The two new channel implementations are `AsyncWebSocketsChannel` and its synchronous counterpart `SyncWebSocketsChannel`. Both are built on the `websockets` library and implement the persistent, full-duplex connection required by the FFP protocol. The existing `AsyncTCPIPChannel` and `SyncTCPIPChannel` retain the behavior of `pyfgc 2.x` and are used for NCRP communication with FGC2 and FGC3 devices. `SerialChannel` provides local serial connectivity for SCRP communication, wrapping the `pyserial` library. `RDACHannel` wraps the `pyrda3` RDA3 client library to provide access to the CMW/RDA stack; unlike the byte-stream channels, it overrides `read()` to return typed structured data (a string or `numpy` array) reflecting RDA’s property-value model, and it does not support the generic message and disconnect callbacks of the other implementations. This non-standard interface is indicated by the yellow fill in fig. 4.3.

A `ChannelManager` class handles the instantiation and lifecycle of channels. It caches channel instances by type and connection parameters, so that multiple connections to the same gateway share a single underlying channel. Channels are created via the manager’s `create()` method and released via `destroy()`, which ensures that network resources are

cleaned up correctly.

4.4. Protocol layer

The protocol layer defines how messages are serialized into bytes for transmission and deserialized from bytes on receipt. The base abstraction is `AbstractProtocol`, which exposes two methods: `process_request(**kwargs) -> AbstractMessage` and `process_response(bytes) -> AbstractMessage`. Concrete protocol classes implement these methods for their specific wire format. Figure 4.4 shows the four implementations.

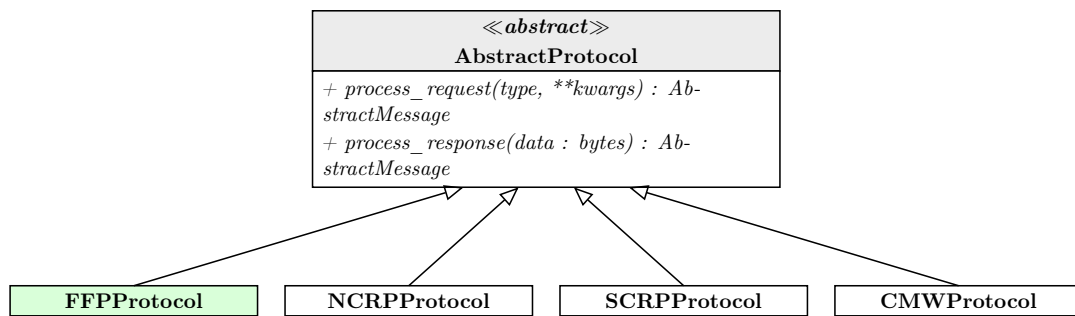


Figure 4.4: Protocol class hierarchy in pyfgc 3.0.0. `FFPProtocol` is the new addition that enables FGC4 communication; the other three protocols support existing device generations. Green fill indicates the class introduced in this version.

4.4.1. FFPProtocol

`FFPProtocol` implements the FGC Front-End Protocol (FFP) [24], a JSON-over-WebSocket protocol designed for FGC4 communication. Requests are JSON-encoded objects transmitted over a WebSocket connection. The protocol defines eleven request types: `INIT` (session initialisation with optional RBAC credentials), `CLEAR` (session reset with optional RBAC credentials), `GET/GETEX` (property read), `SET/SETEX` (property write), `SUB/SUBEX` (property subscription), `UNSUB` (unsubscribe), `TERM` (remote terminal), and `STATUS` (periodic device status). The `EX` variants accept individual fields (device name, property, selector, transaction ID, filter, tag) explicitly, where the compact variants use the command string format described in section 2.3.3. Error reporting uses HTTP status codes for protocol-level errors and a device-specific error code (`FFPCode`) for device-level faults.

4.4.2. NCRPProtocol

`NCRPProtocol` (Network Command Response Protocol) implements the Network Command/Response Protocol used by FGC2 and FGC3 devices. Commands are text strings transmitted over a TCP/IP connection. The protocol is synchronous in nature: each command produces exactly one response. The implementation raises `ProtocolIncompleteMessageError` when the received data does not yet constitute a complete message, allowing the caller to accumulate further bytes.

4.4.3. SCRPProtocol and CMWProtocol

`SCRPProtocol` implements the serial command/response protocol for devices accessed via a serial port. `CMWProtocol` implements the CMW messaging protocol used exclusively with the `RDACHannel` for subscription-based communication with devices on the CERN control system.

This is also an opportunity to correct the naming inherited from `pyfgc` 1.x and 2.x, where the protocol argument accepted the string values `sync`, `async`, `serial`, and `rda` (see listing 3.7). These names describe the transport and execution mode rather than the wire protocol actually in use, and do not match the terminology used by the teams that operate the corresponding services: what was called `sync/async` is NCRP, and what was called `rda` is CMW. `pyfgc` 3.0.0 names each protocol class after the protocol it implements (`FFPProtocol`, `NCRPProtocol`, `SCRPProtocol`, `CMWProtocol`), removing this ambiguity.

All protocol implementations produce and consume instances of `AbstractMessage[M]`, a generic message class that exposes the payload as `.as_bytes`, `.as_dict`, and `.as_json`, providing a uniform interface regardless of the underlying wire format.

4.5. Connection layer

The connection layer composes a channel and a protocol into a typed interface through which the caller exchanges messages. `AsyncConnection[C, P]` is a generic class parameterized over a channel type `C` and a protocol type `P`, declared as Python `TypeVars` bound to `AbstractAsyncChannel` and `AbstractProtocol` respectively. This means `C` and `P` can be substituted with any concrete channel or protocol subclass, while a static type checker still enforces that they implement the abstract base interface; an `AsyncConnection[AsyncWebSocketsChannel, FFPProtocol]` is therefore statically distinguishable from an `AsyncConnection[SerialChannel, NCRPProtocol]`, even though both share the same implementation. Its two primary methods are `send_message(**kwargs)`, which

delegates to the protocol's `process_request` and then writes the resulting bytes to the channel, and `read_response()`, which reads bytes from the channel and delegates to the protocol's `process_response`.

`SyncConnection[C, P]` wraps `AsyncConnection` via `SyncWrapper`, producing a blocking interface with identical semantics. No logic is duplicated: the sync variant delegates every operation to its async counterpart running in a background thread. Figure 4.5 shows the class structure of the layer.

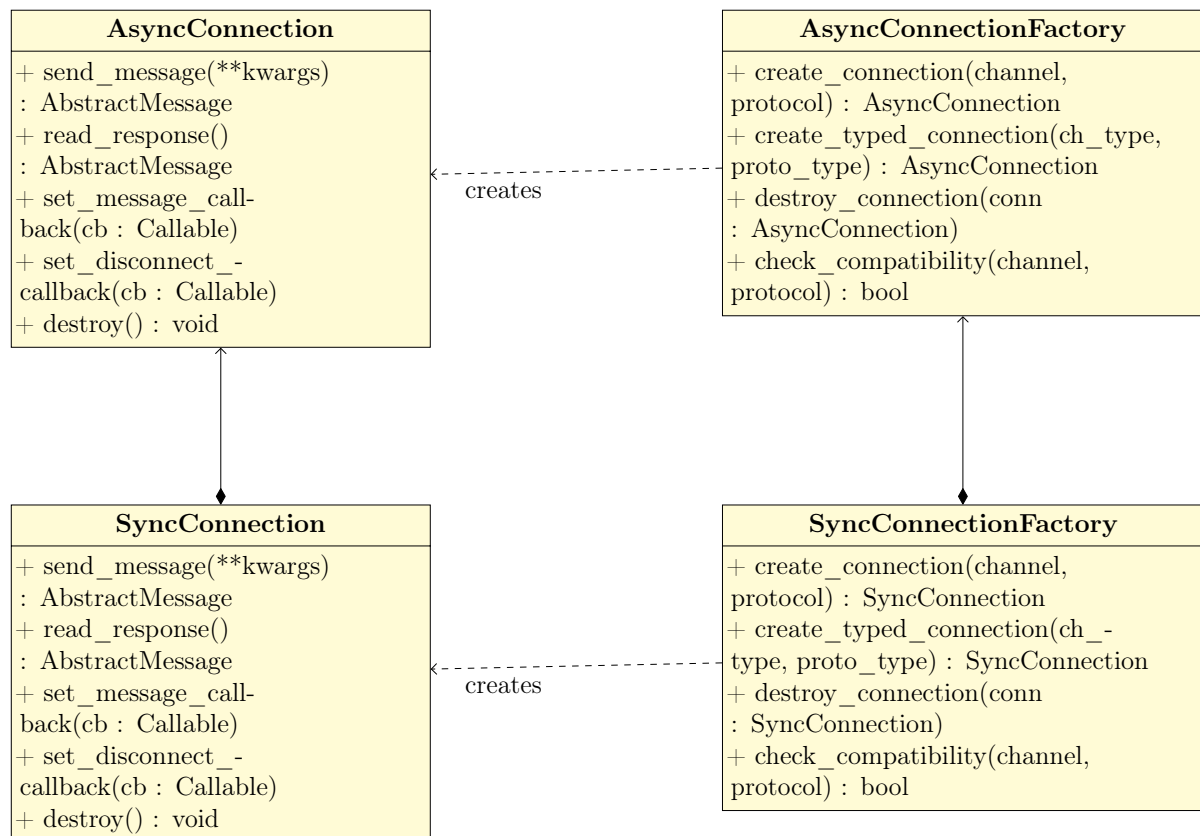


Figure 4.5: Class structure of the pyfgc 3.0.0 connection layer. `AsyncConnection` and `AsyncConnectionFactory` form the async-first implementation; `SyncConnection` and `SyncConnectionFactory` are thin wrappers that delegate every operation to their async counterparts via `SyncWrapper`, providing a fully synchronous interface without duplicating any logic.

The connection supports two callback types. A *message callback* is invoked for every incoming message received on the channel, enabling event-driven communication without polling. A *disconnect callback* is invoked when the underlying transport drops, allowing the caller to respond to connection loss. Both callback types accept either synchronous or asynchronous callables. Command tags, which are optional client-generated strings

attached to requests and echoed in responses, allow reliable request-response matching when multiple requests are in flight concurrently.

Subscription-based communication follows the same pattern but adds an asynchronous dispatch path. Registering a message callback via `set_message_callback` starts an internal background task that continuously reads from the channel and dispatches each incoming message to the registered handler. The client then sends a **SUBEX** message and, once the subscription is acknowledged, receives **NOTIFICATION** messages asynchronously each time the subscribed property changes on the device, without polling. Figure 4.6 shows this flow.

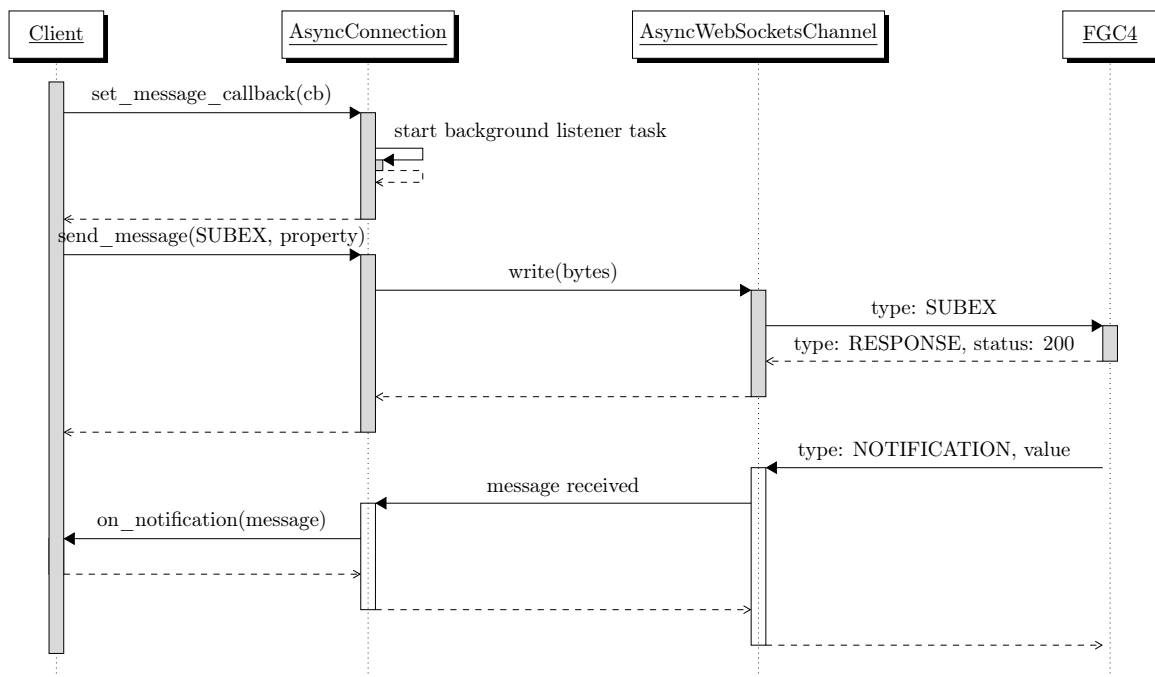


Figure 4.6: FFP subscription flow at the `AsyncConnection` layer. Registering a message callback starts a background task that dispatches every incoming message to the handler. After the **SUBEX** acknowledgement, **NOTIFICATION** messages pushed by FGC4 are received by the background task and delivered to the callback without polling.

Figure 4.7 shows the message flow for an FFP **GET** operation from the connection layer’s perspective.

4.6. AutoConnection layer

The `AutoConnection` module provides a high-level API intended for everyday use by control engineers and script authors who do not need fine-grained control over protocol or channel parameters. It directly addresses the limitation identified in section 3.1.11: in the

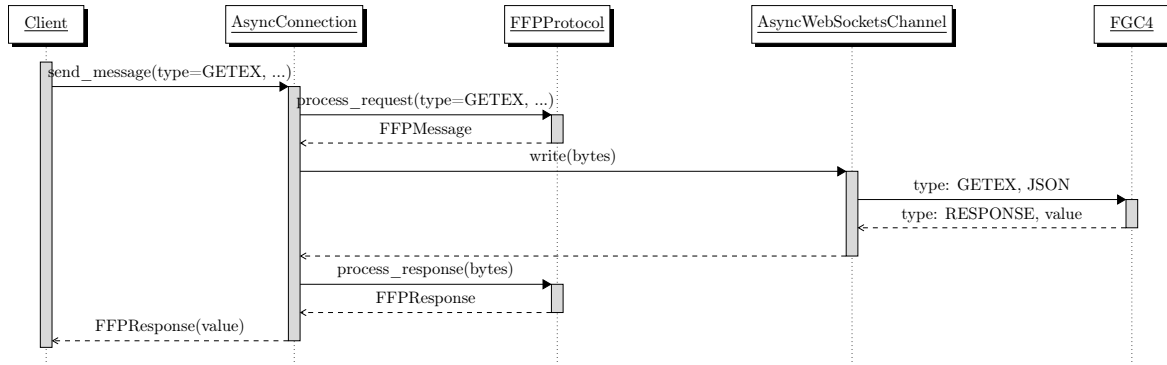


Figure 4.7: FFP GET message flow at the `AsyncConnection` layer. The connection delegates serialization to `FFPProtocol` and transport to `AsyncWebSocketsChannel`. The two layers are fully independent and can be substituted without changing the connection code.

old ecosystem, the caller was responsible for acquiring an RBAC token, locating the name file, and selecting the correct protocol string for the target device generation; `AutoConnection` makes all three automatic. `AsyncAutoConnection` (and its synchronous wrapper `SyncAutoConnection`) exposes two methods: `get(property, ...)` and `set(property, value, ...)`. The response type is `AutoResponse`, which carries `property_value`, `error_code`, `error_message`, and `command_tag`. This unified response type abstracts over the differences between FFP and NCRP response formats. Figure 4.8 shows the class structure of the module.

Connections are obtained through `AsyncAutoConnectionFactory.auto_create_connection()`, which accepts a device name and an optional RBAC token. The factory uses `pyfgc_name` to resolve the device’s gateway address, port, and class, then selects the appropriate channel and protocol pair: `AsyncWebSocketsChannel` with `FFPProtocol` for FGC4 devices, and `AsyncTCPIPChannel` with `NCRPProtocol` for FGC2, FGC3, and FGCLite devices. This selection is automatic and transparent to the caller. Figures 4.9 and 4.10 show the creation and use flow for the FFP (FGC4) and NCRP (FGC3 and earlier) paths respectively.

The `AutoConnection` also handles RBAC token management. The token can be provided at construction time, set or updated after creation, or passed per-request. Before every FFP INIT message, the connection inspects the token’s `.expired` flag; if the token has lapsed, it calls `RBACManager.renew_rbac` in a worker thread via `asyncio.to_thread`, stores the renewed token at the connection level, and proceeds with the INIT message carrying the fresh credentials. The caller receives the property value without handling token expiry. Figure 4.11 shows this flow.

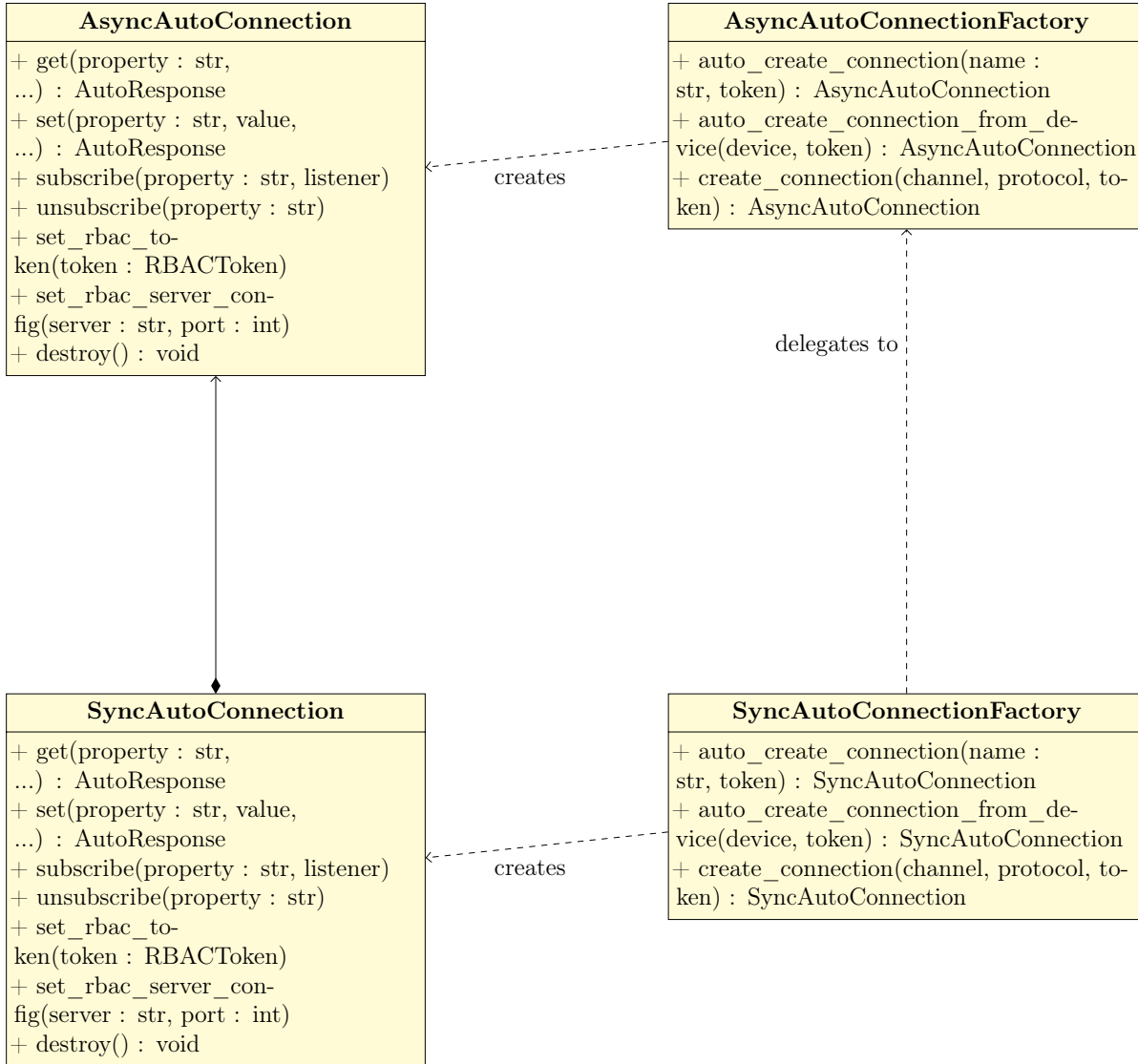


Figure 4.8: Class structure of the pyfgc 3.0.0 AutoConnection layer. **AsyncAutoConnectionFactory** resolves the device name via `pyfgc_name` and selects the appropriate channel and protocol pair (WebSocket with FFP for FGC4, TCP/IP with NCRP for FGC3 and earlier). **SyncAutoConnection** owns an **AsyncAutoConnection** and delegates all operations via **SyncWrapper**; **SyncAutoConnectionFactory** delegates all factory calls to **AsyncAutoConnectionFactory**.

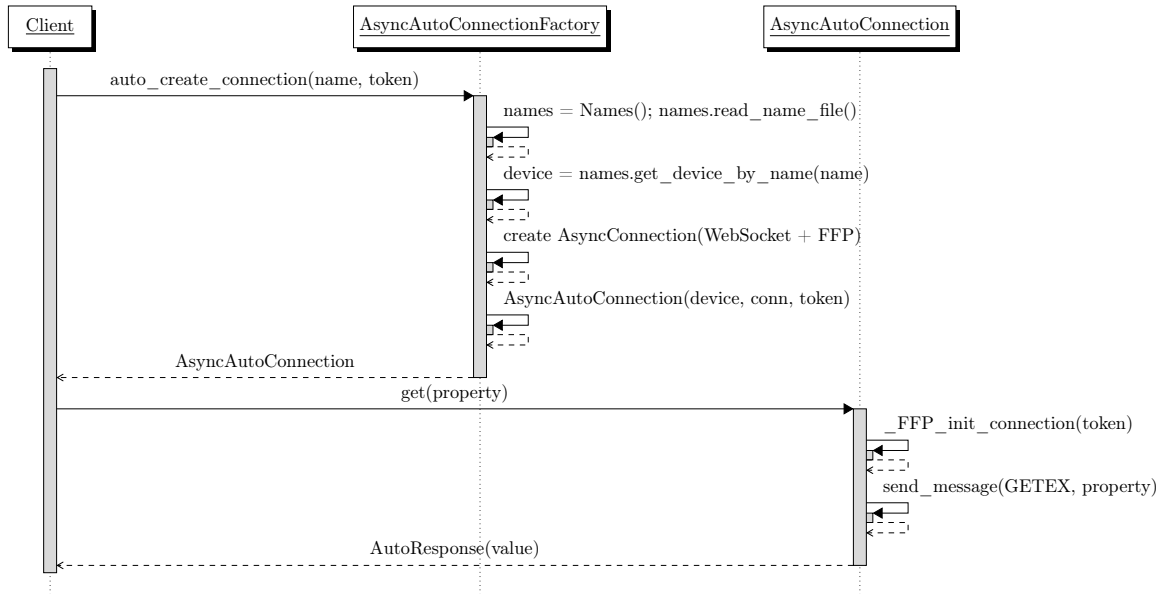


Figure 4.9: AutoConnection creation and use. The RBAC token must be obtained by the caller beforehand via `pyfgc_rbac`; it is passed to the factory and stored on the connection for use in subsequent requests. The factory resolves the device via `pyfgc_name`, selects the appropriate channel and protocol based on the device class, and returns a ready-to-use `AsyncAutoConnection`. The caller interacts only with the high-level `get/set` API.

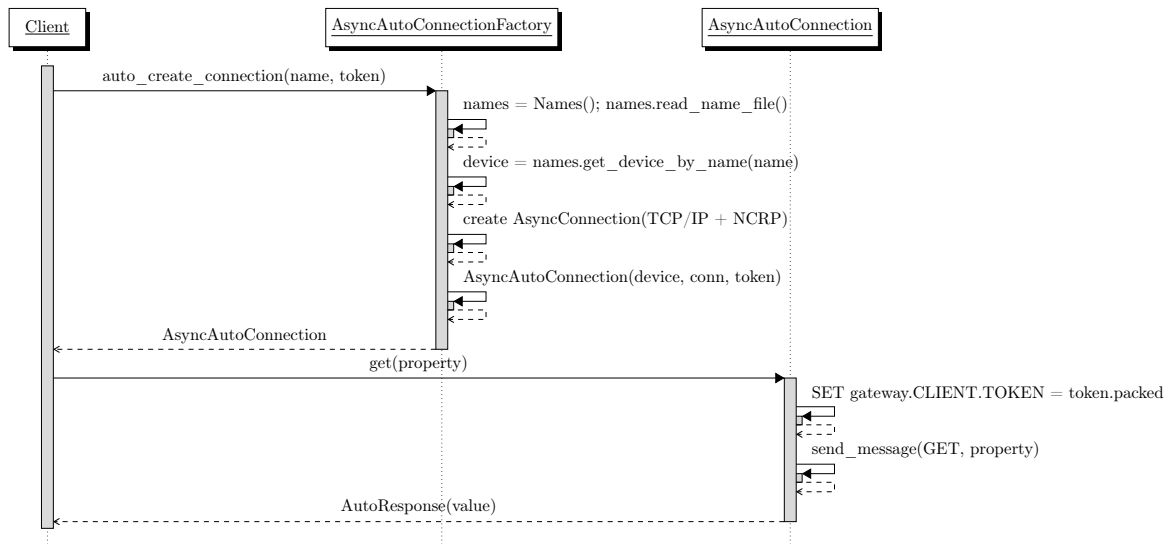


Figure 4.10: AutoConnection creation and use for an FGC3 device. The factory selects `AsyncTCPIPChannel` with `NCRPProtocol` based on the device class. Authentication differs from the FFP path: instead of an INIT handshake, the RBAC token is written to the `CLIENT.TOKEN` property on the gateway before the `GET` request is issued on the device.

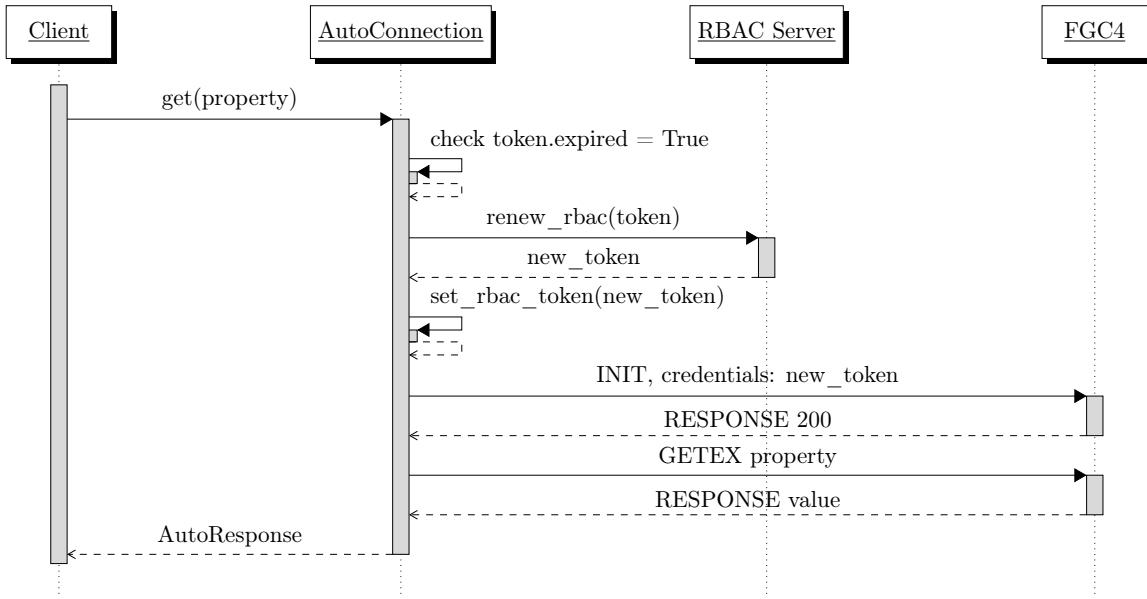


Figure 4.11: Automatic RBAC token renewal in `AutoConnection`. When an expired token is detected before the FFP INIT handshake, the connection transparently renews it via the RBAC server and attaches the fresh credentials to the handshake. The caller receives the property value without handling token expiry.

If the underlying transport drops, the disconnect callback notifies the caller.

For applications that create many connections to different devices, a `Names` instance can be pre-loaded and injected into the factory, avoiding repeated name file fetches. Alternatively, a pre-resolved `Device` object can be passed directly via `auto_create_connection_from_device()`.

4.6.1. CMW/RDA subscription support

A subsequent extension (EPCTOOLS-292) adds property subscription support to the `AutoConnection` layer via the CMW/RDA protocol stack [31]. Unlike FFP and NCRP, which follow a request/response model, CMW/RDA is a publish/subscribe protocol: a registered listener receives updates whenever a monitored property value changes on the device, without polling.

Subscription support is activated by passing `use_rda=True` to `AsyncAutoConnectionFactory.auto_create_connection()` or its synchronous counterpart. This flag instructs the factory to create an `RDChannel`-based connection regardless of the device class returned by `pyfgc_name`. A valid RBAC token is mandatory when `use_rda=True`. Once the connection is obtained, `subscribe(property, listener)` registers an `RDChannelListener`

to receive updates, and `unsubscribe(property)` removes it. The listener declares two callbacks: `data_callback`, invoked on each successful update, and the optional `error_callback`, invoked on subscription errors; if not overridden, errors are forwarded to the message callback. An optional `cycle_selector` argument filters delivered updates to a specific accelerator timing cycle. Because the CMW/RDA infrastructure is available only within CERN's network, subscriptions via this path are not supported for FFP, NCRP, or SCRP connections.

4.7. Real-time data

The `rt_data` module supports sending real-time current reference values to FGC gateway devices via UDP. This is the mechanism used to update converter current references at high frequency, outside the request/response model. `RealTimeData` holds the reference values for up to 64 fieldbus channels; each floating-point value is encoded as an unsigned 32-bit IEEE-754 bit pattern. `RealTimeConnection` is initialized from a gateway `Device` and maps each sub-device to its fieldbus channel number via the `fieldbus_address` attribute. The `send_refs()` method builds a binary UDP packet containing the sequence number, send and apply timestamps, the active-channel bitmask, and all reference values, then dispatches it over a non-blocking UDP socket to the gateway. An optional apply timestamp allows scheduling reference updates for a future cycle. `RealTimeConnectionFactory` resolves a gateway device from a name file and returns a ready-to-use `RealTimeConnection`.

4.8. Utilities

The `utils` module contains the `SyncWrapper` class, which is the mechanism by which all synchronous APIs are generated from their asynchronous counterparts. `SyncWrapper` maintains a dedicated `asyncio` event loop running in a background daemon thread. When a synchronous caller invokes a wrapped method, `SyncWrapper.run_sync(coro)` schedules the coroutine on the background loop and blocks the calling thread until the result is available. The helper `safe_wait()` transparently handles both awaitable and non-awaitable return values, so that async methods which happen to return immediately do not require special treatment.

This design means that `SyncConnection` contains no logic of its own beyond forwarding calls to `AsyncConnection` via `SyncWrapper`. The same applies to every other synchronous class in the library: `SyncAutoConnection` and its factory forward to `AsyncAutoConnection`, and the synchronous channel implementations (`SyncTCPIPChannel`, `SyncWeb-`

`SocketsChannel`, and the synchronous channel manager) forward to their asynchronous counterparts in the same way. Every bug fix, new feature, or callback addition needs to be implemented only once, in the async code path. Figure 4.12 illustrates the dispatch sequence.

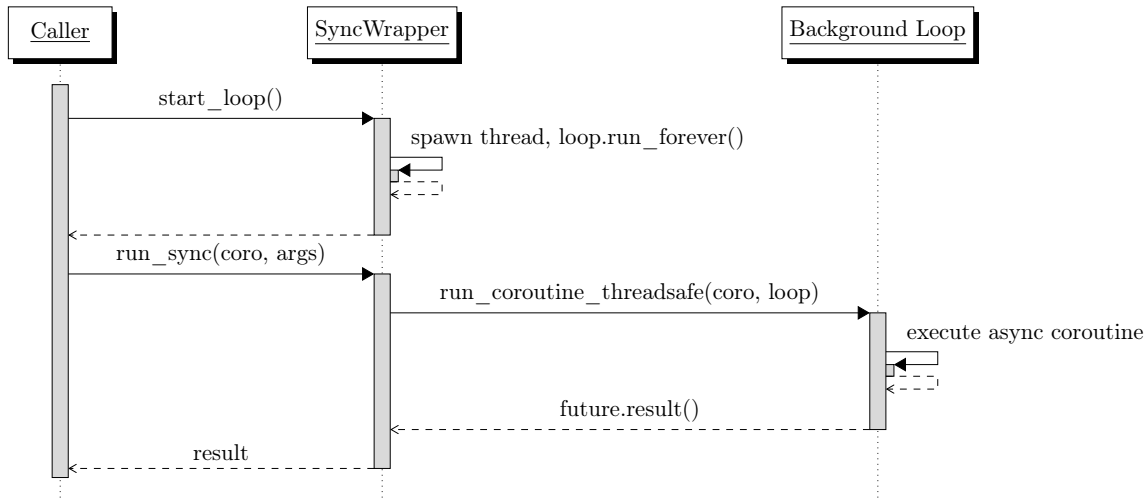


Figure 4.12: `SyncWrapper` dispatch. Starting the wrapper once spawns a daemon thread running a dedicated `asyncio` event loop. Each `run_sync()` call submits a coroutine to the background loop via `run_coroutine_threadsafe`, blocks the calling thread on the future result, and returns the value once the coroutine completes.

4.9. Companion packages

4.9.1. `pyfgc_name`

The `pyfgc_name` package provides device name resolution: given a device name such as `RFNA.866.01.ETH1`, it returns the gateway host, device class, fieldbus address, and other parameters needed to open a `pyfgc` connection. Version 3.0.0 replaces the module-level global dictionaries of the previous version with an object-oriented design centered on the `Names` class.

A `Names` instance is created by the caller and populated by calling `read_name_file()`, optionally followed by `read_sub_device_file()` and `read_group_file()`. The name files are plain-text colon-separated records. The default paths are URLs hosted on a CERN-internal web server that mirrors the NFS share where the live name files reside, so no NFS mount is required on the client machine; callers on machines with the share mounted may instead pass the corresponding local path constants defined in `NameFiles`. The library dispatches transparently: a path beginning with `http` is fetched via an HTTP

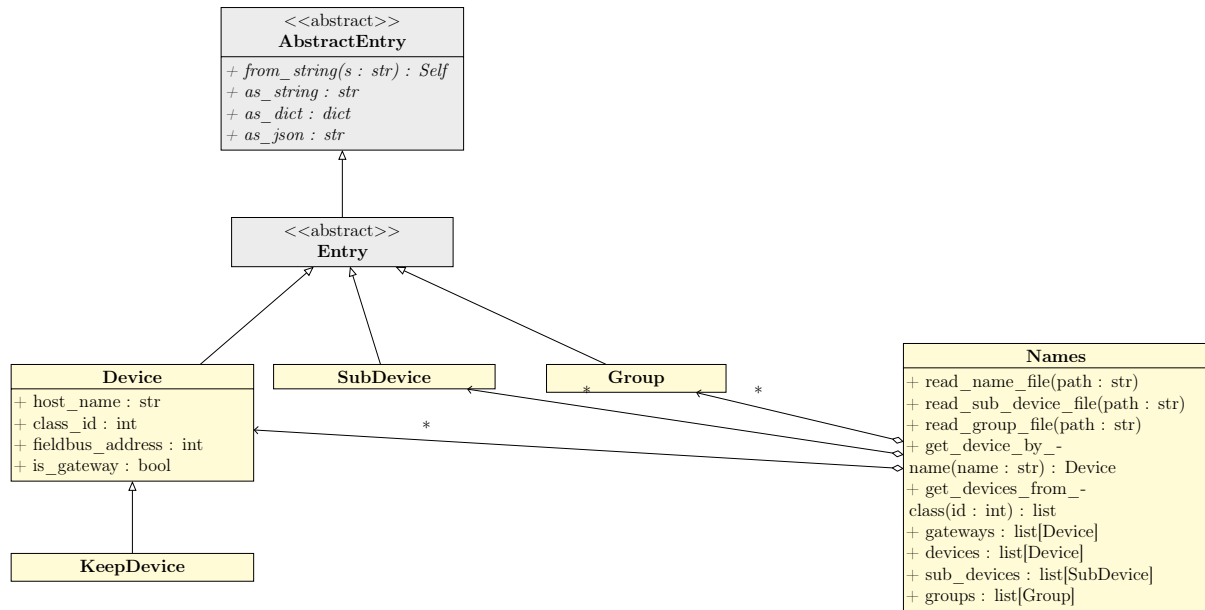


Figure 4.13: Class structure of `pyfgc_name` 3.0.0. `AbstractEntry` defines the common serialization interface; `Entry` provides a partial implementation; `Device`, `SubDevice`, and `Group` are concrete entity classes. `KeepDevice` represents devices in the *keep* operational category. `Names` aggregates entity instances and exposes loading and query methods.

request; any other string is opened as a local file path. All parsed entities are represented by dedicated classes that inherit from `AbstractEntry`: `Device` (with attributes `host_name`, `fieldbus_address`, `class_id`, `name`, and `omode_mask`), `SubDevice`, `KeepDevice` (for devices in the *keep* operational category), and `Group`. Gateways are identified by a fieldbus address of zero; all other devices carry a non-zero address and hold a direct `parent_gateway` reference to their gateway `Device` object. A device's *alias* is derived from the name of its first sub-device (index 0), allowing the library to expose the user-facing short name separately from the internal device name. Each entity class exposes `.as_dict`, `.as_string`, and `.as_json` for serialization. Because all state is held in the `Names` instance, multiple independent instances can coexist in the same process without interference, and unit tests can inject custom name data without touching any shared state.

4.9.2. `pyfgc_rbac`

The `pyfgc_rbac` package handles acquisition and lifecycle management of CERN RBAC tokens, which are required for authenticated access to FGC devices. Version 3.0.0 replaces the module-level functions and global configuration variables of version 1.x with two classes: `RBACToken` and `RBACManager`. This addresses the two limitations identified in

section 3.1.5 and section 3.1.7: the absence of a structured token type meant that expiry information was inaccessible without manual payload decoding, and the dictionary returned by `token_to_dict()` was completely untyped and undocumented, leaving callers to infer field names and value types from the token payload directly, and renewal logic had to be duplicated by every consumer.

`RBACToken` is an immutable representation of a single token. It decodes the token payload at construction time and exposes typed properties: `.expired` (boolean), `.seconds_until_expiration` (float), `.as_bytes` (raw bytes for storage or transmission), `.as_base64` (string for JSON payloads such as FFP INIT messages), and `.packed` (the encoding expected by the NCRP `CLIENT.TOKEN` property). The `verify()` method validates the token's RSA signature against CERN's public key. Dictionary-style access (`token["roles"]`) is supported for inspecting token fields without requiring callers to decode the payload themselves.

`RBACManager` provides the authentication methods that return `RBACToken` instances: `get_location()` for trusted-network location-based authentication, `get_kerberos()` for Kerberos/GSSAPI single sign-on (an addition in this version), `get_explicit()` for username/password login, `get_ssh()` for retrieving a token from a remote machine via SSH, `get_saml()` for SAML response exchange, and `get_oauth2()` for OAuth2 token exchange. Tokens can be persisted to disk with `store_rbac()` and reloaded with `load_rbac()`, and renewed without re-authenticating via `renew_rbac()`. `reduce_rbac_roles()` produces a token with a restricted role set when least-privilege access is required. Because `RBACManager` is instantiated by the caller, different instances can be configured with different server addresses, enabling use outside the CERN network.

4.9.3. `pyfgc_statussrv`

The `pyfgc_statussrv` package provides a client for the FGC status server, a service of class `FGCD_STATUS` that periodically broadcasts binary status frames containing the operational state of all devices on a gateway. The status server device itself is of class `FGCD_EQ_STATUS` and is named `FGC_STATUS` by convention. Version 2.0.0 replaces the previous monolithic module with two classes that separate the concerns of data retrieval and binary decoding.

`StatusServerClient` exposes three static methods. `get_status(connection, device_name)` accepts a `pyfgc AsyncConnection` and issues an NCRP GET for the ALL property on the status server device, accumulating response chunks until the frame terminator `n;` is detected. A guard against false positives additionally requires the data section

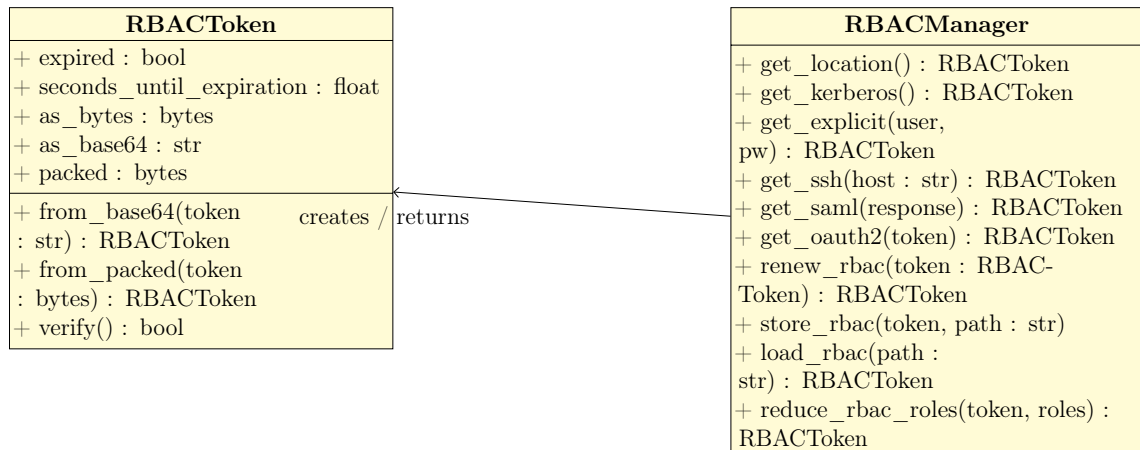


Figure 4.14: Class structure of `pyfgc_rbac` 3.0.0. `RBACToken` is an immutable value object wrapping a decoded RBAC token; all authentication methods on `RBACManager` return `RBACToken` instances. `from_base64` and `from_packed` are static factory methods. `get_kerberos()` is new in this version.

length to be a non-zero multiple of `FGC_STATSRV_FGCD_SIZE`, since the byte sequence `0x0a 0x3b` can appear inside binary device data. `get_status_from_channel(channel, device_name)` accepts a raw `AsyncTCPIPChannel` for lower-level access, and `decode_data(response)` performs decoding only from already-received bytes.

`StatusDecoder` handles the binary parsing of status frames. The decoding logic was preserved substantially from the previous version and only lightly refactored for the new architecture. `decode_status()` skips the 9-byte NCRP header, then iterates through the response in fixed-size `FGC_STATSRV_FGCD_SIZE` blocks. For each block, `decode_header()` extracts the gateway hostname and the list of device names, and `decode_device_block()` decodes each device’s binary record using the class-specific decoder from `pyfgc_decoders`, falling back to the common decoder when no class-specific decoder is found. The returned dictionary is indexed by gateway hostname, with each entry containing a `devices` sub-dictionary mapping device names to their decoded status fields. Separating the decoder from the client makes it straightforward to unit-test the binary parsing logic independently of network communication.

4.9.4. `pyfgc_decoders`

The `pyfgc_decoders` package contains binary decoders for the data types used in FGC status and property responses. The package predates this work; the contributions in version 0.0.12 are additions of type annotations to all public functions, additional unit tests covering previously untested edge cases, and the introduction of structured logging

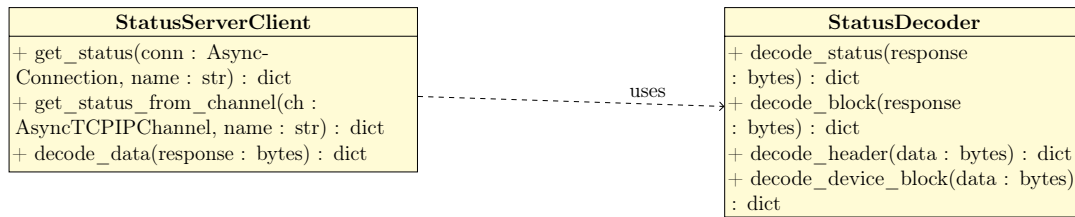


Figure 4.15: Class structure of `pyfgc_statussrv` 2.0.0. All methods on both classes are static. `StatusServerClient` handles network communication and delegates binary decoding to `StatusDecoder`. The dashed arrow indicates a dependency: `get_status()` and `get_status_from_channel()` call `StatusDecoder.decode_status()` internally.

to aid debugging of decoding failures.

4.9.5. `ccs-fastapi-utils`

The `ccs-fastapi-utils` package provides FastAPI middleware and utilities for CERN web services that require RBAC authentication. Version 1.0.15 updates the package to use the new `RBACToken` class introduced in `pyfgc_rbac` 3.0.0, replacing direct manipulation of token bytes and dictionaries with the typed interface. A bug in the RBAC authentication middleware that caused valid tokens to be incorrectly rejected under certain role combinations was corrected. Type annotations were added throughout, and structured logging was introduced for authentication events.

4.9.6. `pyfgc_test_framework`

The `pyfgc_test_framework` package provides utilities for writing integration tests that exercise FGC devices: wrappers around get/set operations, timing helpers, and an `authenticate` module that manages RBAC token acquisition during test runs. The package predates this work and was not modified by the author; however, it was updated by a colleague (EPCTOOLS-283) to be compatible with `pyfgc` 3.0.0 as part of the broader ecosystem migration.

The `authenticate` module was updated to replace the previous pattern of calling module-level functions (`pyfgc_rbac.get_token_kerberos()`, `pyfgc_rbac.is_expired()`) with the typed `RBACToken` class and an `RBACManager` instance from `pyfgc_rbac` 3.0.0. The `boot` module, which communicates with the FGC boot program, was updated to use `StatusServerClient` from `pyfgc_statussrv` instead of the removed `pyfgc` functions it previously relied on. Test cases were updated to catch `AutoConnectionError` in place of the removed `pyfgc.FgcResponseError`, and the response inspection pattern `res.err_`

`code / res.err_msg` was updated to `res.has_error / res.error_message` to match the `pyfgc 3.0.0` response API.

4.10. Web application integration

4.10.1. FRESCO

FRESCO is a web application for automated experimentation on FGC devices, described in section 2.6.2. At the time of writing, FRESCO has not been migrated to `pyfgc 3.0.0` and continues to use `pyfgc 1.8.6` for device communication. A migration is planned as a follow-up activity. The application relies on `pyfgc` to open and manage FGC sessions, read and write device properties, and authenticate via RBAC tokens; a future migration would follow the same pattern applied to the FGC API and PowerSpy, replacing direct use of `pyfgc.connect` with `AsyncAutoConnectionFactory.auto_create_connection()`.

4.10.2. FGC Commander

FGC Commander is the web application for monitoring and commanding FGC devices from a browser, described in section 2.6.2. Its backend is responsible only for user-specific metadata (device tags, device class configuration, and permalinks); all FGC device communication is delegated to the FGC API. As a result, the FGC Commander backend does not use the core `pyfgc` package for device access: it uses only `pyfgc_rbac` for token validation in its authentication middleware, through `ccs-fastapi-utils`.

FGC Commander was not migrated as part of this work. Because its device interaction is fully delegated to the FGC API, which was migrated in section 4.10.3, FGC Commander benefits indirectly: all commands and monitoring data now flow through an FGC API instance backed by `pyfgc 3.0.0` and FFP, without any changes to FGC Commander itself.

4.10.3. FGC API

The FGC API, described in section 2.6.2, is a FastAPI-based REST web service that provides a centralized interface for accessing FGC configuration data and sending commands to devices. At the start of this work, the service was at version 1.10.0 and used `pyfgc 2.x` for device communication via the NCRP protocol.

Version 2.0.0 migrates the FGC API to `pyfgc 3.0.0`. The most significant architectural change is the removal of the queue-based gateway connection pool (`FgcPoolWrapper`), which serialized requests to each gateway through a shared queue. Under the old model,

all device commands destined for the same gateway were enqueued and dispatched sequentially, limiting throughput and introducing latency under load. Furthermore, because all connections were established at startup and held open for the lifetime of the process, a device reboot during normal operation would leave FGC API unable to reconnect, requiring a full service restart to recover. In the new design, each request creates a direct connection through `AsyncAutoConnectionFactory`, which leverages the async-first architecture of `pyfgc 3.0.0` to handle concurrent in-flight requests natively. Since connections are created on demand, a rebooted device is transparently reached on the next request with no additional overhead. Beyond correctness and throughput, the per-request connection model also simplifies the codebase significantly: the queue management and pool bookkeeping logic is removed entirely, reducing both code complexity and CPU usage. Figure 4.16 illustrates the per-request connection lifecycle, and Figure 4.17 shows the resulting server architecture.

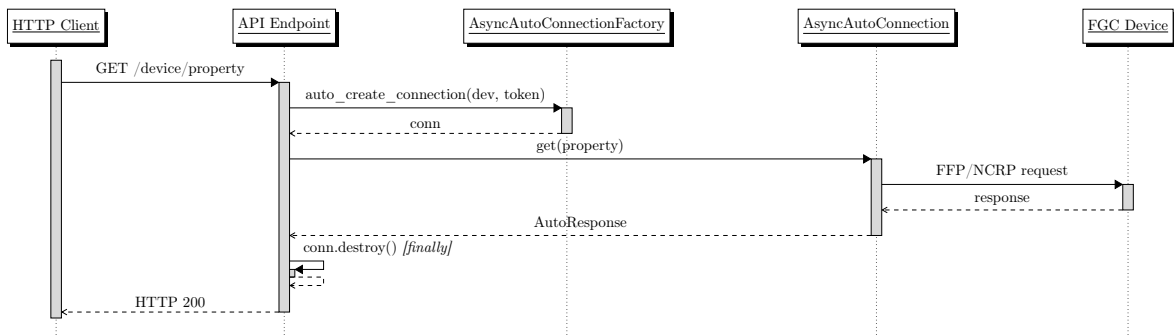


Figure 4.16: Request handling in FGC API 2.0.0. Each incoming HTTP request creates a dedicated `AsyncAutoConnection` through `AsyncAutoConnectionFactory` (see fig. 4.9) and destroys it in a `finally` block after the operation completes. This per-request connection model replaces the queue-based `FgcPoolWrapper` of version 1.x, allowing concurrent in-flight requests to different devices without serialisation.

The configuration module was refactored from a single flat `Config` class into a hierarchical structure composed of four focused sub-configurations (application settings, authentication, database, and operational files), each extending `CcsFastApiConfig` from `ccs-fastapi-utils`. This allows CERN-specific configuration (OAuth endpoints, RBAC audience, external lab name) to be isolated in a dedicated sub-class, making the service easier to deploy in both CERN and non-CERN environments.

The authentication module was updated to use the `RBACToken` class from `pyfgc_rbac 3.0.0`, replacing the previous pattern of calling module-level functions (`pyfgc_rbac.set_rbac_server()`, `pyfgc_rbac.token_to_dict()`) with the typed token interface. Dedicated CI pipeline jobs for linting and testing were added to the GitLab CI configuration. The

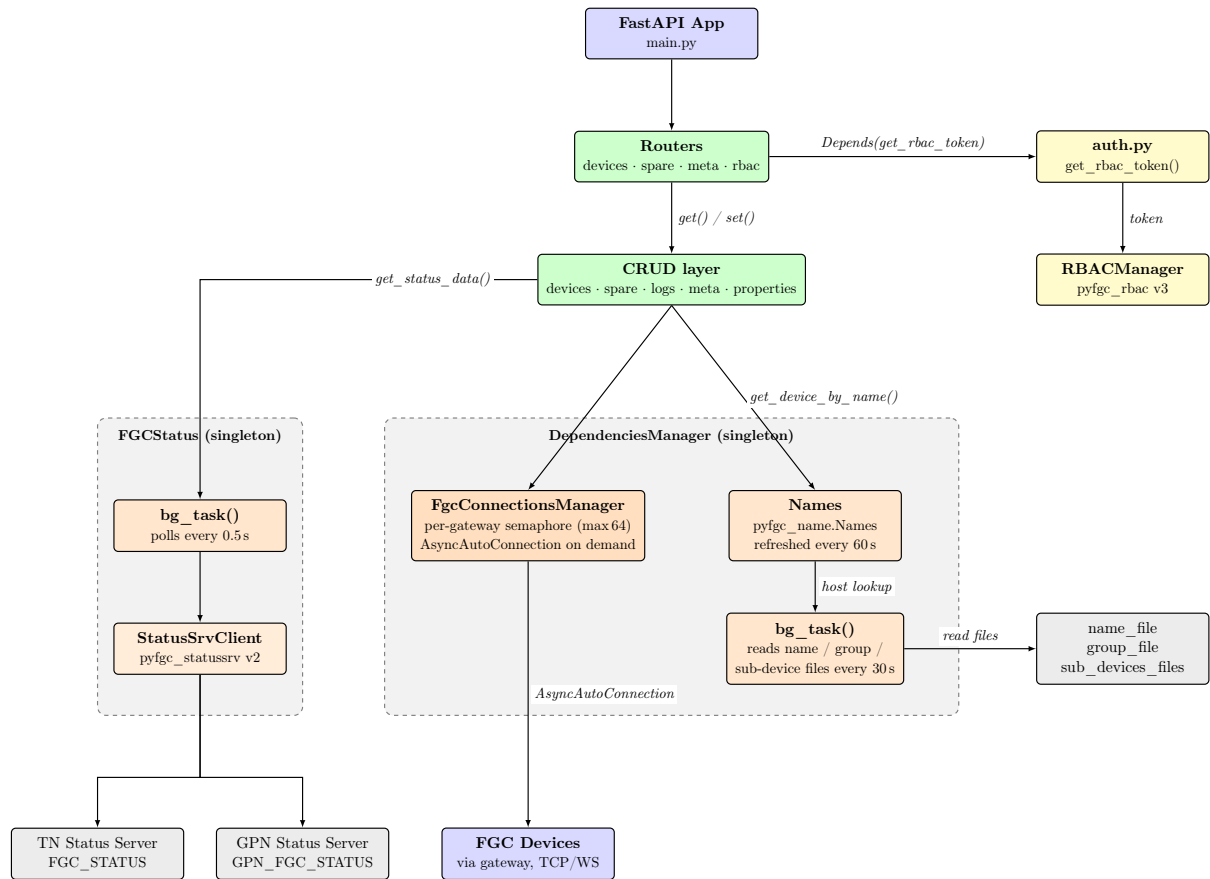


Figure 4.17: Server architecture of FGC API 2.0.0. The FastAPI application exposes HTTP endpoints through a router layer, which delegates device operations to a CRUD layer. A DependenciesManager singleton owns a Names instance (refreshed every 60s by a background task reading the CERN name files every 30s), an RBACManager, and an FgcConnectionsManager that creates AsyncAutoConnection instances on demand bounded by a per-gateway semaphore of 64. A separate FGCStatus singleton polls device status via pyfgc_statussrv every 0.5s from both the TN and GPN status servers. The GPN instance exists because development FGC4 boards reside on the GPN; relocating them to the TN would require physical switch migration, increased security procedures, and IT oversight of the lab. Since FGC4 firmware introduces support for publishing device status to the status server, a dedicated FGCD_STATUS instance was deployed on the GPN so that development FGC4 boards can publish their status without requiring TN access.

Dockerfile was updated to use `uv` for dependency resolution, replacing the previous `pip` workflow with a locked requirements file generated from `uv.lock` at build time. `uv` is a Python package and project manager that resolves and installs dependencies significantly faster than `pip` and produces a `uv.lock` lockfile pinning the exact version of every dependency, including transitive ones, for reproducible installations; the Dockerfile uses this lockfile to generate a pinned requirements file at build time, ensuring that the image's dependency set is reproducible across builds.

4.10.4. PowerSpy

PowerSpy is a web application for visualizing real-time and offline data from FGC devices and SVCs. At the start of this work, the server-side controller used `pyfgc` 2.x through its module-level asynchronous API (`pyfgc.async_get`, `pyfgc.async_fgc` context manager) and resolved device names through the global dictionary `pyfgc_name.devices`.

The migration to `pyfgc` 3.0.0 replaces all direct calls to the old module-level API with explicit `AsyncAutoConnection` objects created through `AsyncAutoConnectionFactory.auto_create_connection()`. A new `create_connection()` helper was introduced in the controller module to centralize connection creation: it instantiates a `Names` object, loads the name file, and delegates to the factory. Each operation (timing query, log menu fetch, log acquisition, multiplexer configuration) now creates a connection at the start of the call and explicitly destroys it in a `finally` block upon completion.

The field `err_code` on FGC response objects, which was renamed to `error_code` in `pyfgc` 3.0.0, was updated at all call sites. The authentication module was updated to replace the global `pyfgc_rbac.set_rbac_server()` initialization with an `RBACManager` instance, consistent with the new `pyfgc_rbac` 3.0.0 API. The dependencies in `pyproject.toml` were updated to require `pyfgc` 3.0.0, `pyfgc_name` 3.0.0, `pyfgc_rbac` 3.0.0, and `pyfgc-decoders` 0.0.12.

4.11. DevOps and CI/CD extensions for testing

4.11.1. Development methodology

Development at SY-EPC-CCS is organized using an Agile sprint methodology. Work is tracked in JIRA, where each unit of work corresponds to a ticket with a project code and number (e.g., FGC-API-13 for the `pyfgc` migration in the FGC API, or POWERSPY-98 for the corresponding PowerSpy migration). Sprints are time-boxed periods of development,

and each sprint has a dedicated integration branch in the GitLab monorepo following the naming convention `sprint/project-YYYY-MM-context` (e.g., `sprint/epctools-2026-03-epcall`).

For each JIRA ticket, a feature branch is created from the sprint branch following the convention `dev/username/ticket-id-description`. Development proceeds on the feature branch; when the work is complete, a merge request is opened against the sprint branch. Merge requests trigger the CI pipeline (linting, testing, quality analysis), and the code is reviewed by at least one other team member before being merged. At the end of the sprint, the sprint branch is merged into the default branch (`master`), triggering the build and deployment pipeline.

The software developed in this work is distributed across two GitLab monorepos, each with a distinct CI/CD strategy matched to the nature of its artifacts: the `fgc` repository [29] contains Python packages, while the `fgc-web` repository [28] contains web applications deployed as Docker containers. Both monorepos host additional software beyond the scope of this work; only the packages and applications described in this chapter are discussed here.

4.11.2. Python package pipeline

All Python packages (`pyfgc`, `pyfgc_name`, `pyfgc_rbac`, `pyfgc_statussrv`, `pyfgc_decoders`, and `ccs-fastapi-utils`) reside in subfolders of the `fgc` monorepo and share a single root `.gitlab-ci.yml` that aggregates per-package job files. Each package defines its own `.gitlab-ci.jobs.yml`, included at the root level, which extends shared job templates from the ACC-Py CI template library [7] (see section 2.6.3).

The pipeline distinguishes two triggers: code changes on any branch, and package-specific version tags. On a code change, the pipeline builds a source distribution and a wheel, runs the unit test suite in a development installation and then in a clean wheel installation, and publishes a development snapshot of the documentation. On a tag matching a package-specific pattern (e.g., `pyfgc-v3.0.0` for the `pyfgc` package), the same build and test jobs run, the release documentation is published, and the `publish` job uploads the wheel and source distribution to the ACC-Py package index, making the new version immediately installable via `pip`.

Version numbers are managed automatically using `setuptools-scm`. The `version` field in `pyproject.toml` is declared `dynamic`, and `setuptools-scm` derives the version at build time from the most recent Git tag matching the package-specific tag regex (e.g., `^pyfgc-v(?P<version>.+)$`). This eliminates manual version bumps in source files: pushing a

Git tag of the form `pyfgc-v3.0.0` is the only action required to cut a release.

4.11.3. Web application pipeline

The `fgc-web` monorepo hosts all the web applications and follows a GitOps deployment model. The root `.gitlab-ci.yml` aggregates per-application job files through six pipeline stages: lint, build, test, quality, deploy, and smoke.

Lint and test jobs run only on merge request events when the relevant files have changed, keeping feedback fast for reviewers. Server tests run against a real PostgreSQL instance provided as a GitLab CI service, avoiding the mock/production divergence that can mask integration failures. Client tests use Cypress. After all test jobs complete, a SonarQube analysis job collects coverage reports from all applications and evaluates the results against a shared quality gate; the job is non-blocking but its results are visible on the merge request.

Build jobs run on merges to `master`: Kaniko builds Docker images for both the Python server and the Vue.js client, and the resulting images are pushed to the CERN container registry (`registry.cern.ch/ccs-tools/`) tagged with the commit SHA. Immediately after a successful build, the deploy job patches the new image SHA into the `latest.yaml` tracking file and pushes the change back to `master`. ArgoCD, configured to poll the repository for changes, detects the updated manifest and reconciles the live *dev* deployment accordingly, with no further manual action required.

Two named target environments are defined, each with its own values file under `deployment/kubernetes/.envs/`: *dev* and *prod*. The CI pipeline automatically keeps the *dev* environment in sync with the latest master build through ArgoCD. Promotion to *prod* is a manual step, performed using the `k8s-deploy.sh` wrapper script, which tags the selected image version, waits for the container registry to confirm the image is available, and updates the target environment's values file.

Each application is deployed as a Kubernetes workload described by a Helm chart. Secrets are encrypted at rest using SOPS with GPG and decrypted at deploy time. An NGINX ingress controller handles TLS termination (certificates issued by Let's Encrypt via cert-manager) and path-based routing between applications. The FGC API is deployed with sticky sessions enabled: subsequent requests from the same client are routed to the same replica, ensuring that the per-replica gateway connections established by `pyfgc 3.0.0` are reused rather than re-created on every request.

4.11.4. Testing infrastructure

Both pipelines treat automated testing as a priority. In the Python package pipeline, every merge request triggers the full unit test suite against both a development installation and a clean wheel installation, ensuring that packaging artefacts are themselves tested rather than only the source tree. The test jobs run under Python 3.14 on the ACC-Py CI runners and produce coverage reports that feed the quality gate. In the web application pipeline, server-side tests run against a live PostgreSQL instance provided as a GitLab CI service rather than against a mock, and client-side tests use Cypress to exercise the full browser rendering path. SonarQube collects combined coverage data from all applications and evaluates them against a shared quality gate on every merge request.

These pipeline investments directly support the validation strategy described in chapter 5: the 946-test unit suite runs automatically on every code change, and integration test results against real FGC hardware are collected in the same CI environment. Together, the pipelines form a continuous feedback loop that prevents regressions across the full stack as the FGC4 deployment evolves.

4.12. Contributions to the FGCDv2 gateway firmware

In addition to the Python ecosystem described in the preceding sections, this work includes a set of contributions to the `fgc-next` repository [30], the C++ monorepo that hosts the FGCDv2 gateway daemon running on FGC4 hardware. FGCDv2 is the server-side counterpart of the Python FFP client stack: it receives FFP requests from connected clients, dispatches them to the device command layer, and sends back responses and subscription notifications. These contributions address four distinct concerns: the addition of a C++ FFP client class, the introduction of a standardised error-handling vocabulary, the removal of a legacy integer type library, and the cleanup of the FFP protocol interface headers.

Each FGC4 installation is organised as a cluster: one gateway board runs FGCDv2 and exposes the FFP WebSocket server to the network, while up to three slave boards handle the actual I/O and fast control operations. External clients communicate exclusively with the gateway; the gateway in turn communicates with the slaves over an internal fieldbus. Figure 4.18 shows where the FGC4 boards and FGCDv2 fit within the broader CERN control system, and fig. 4.19 shows the deployment topology of FGCDv2 on the FGC4 hardware.

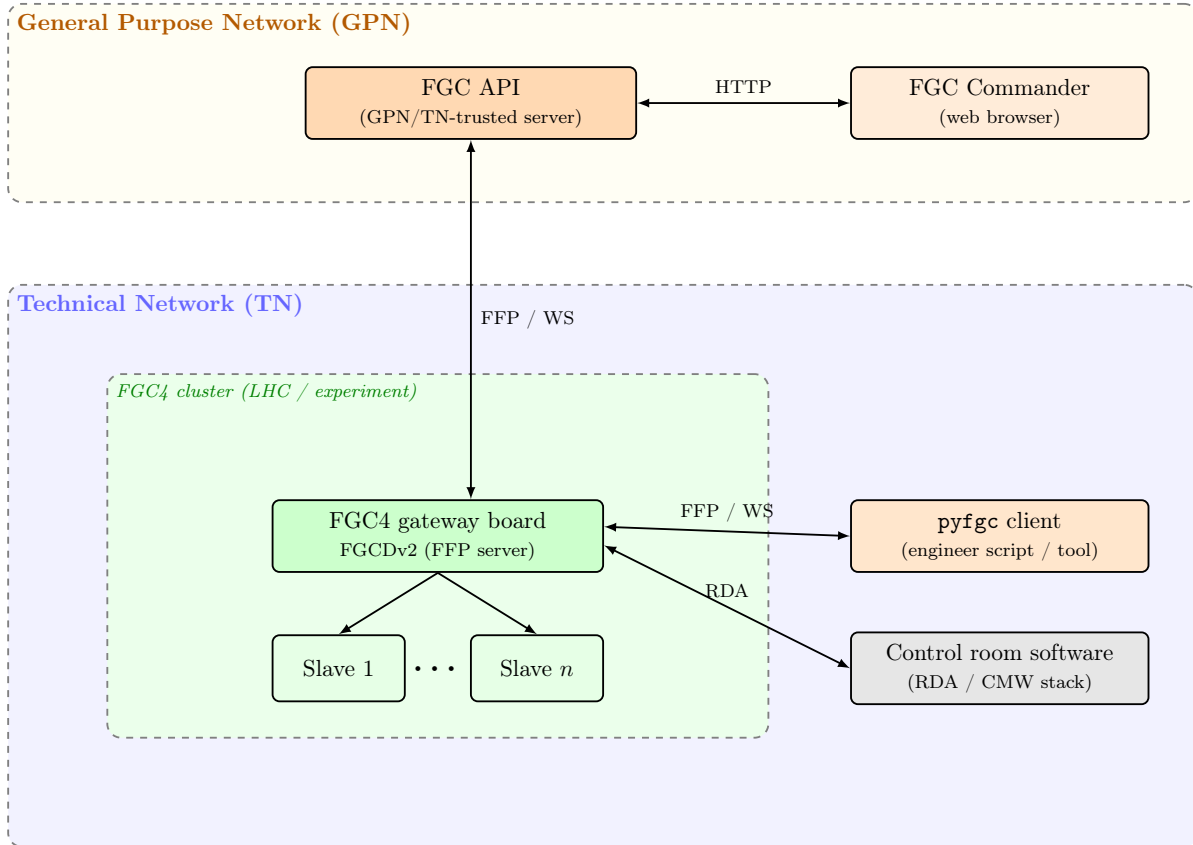


Figure 4.18: LHC control system context for FGC4 and FGCDv2. FGC4 boards reside in the Technical Network (TN); each cluster consists of one gateway board running FGCDv2 (the FFP WebSocket server) and up to three slave boards. Clients in the TN communicate via FFP directly using `pyfgc` or the RDA/CMW stack. The FGC API runs on a machine with access to both the TN and the GPN, bridging browser-based tools such as FGC Commander to the device layer.

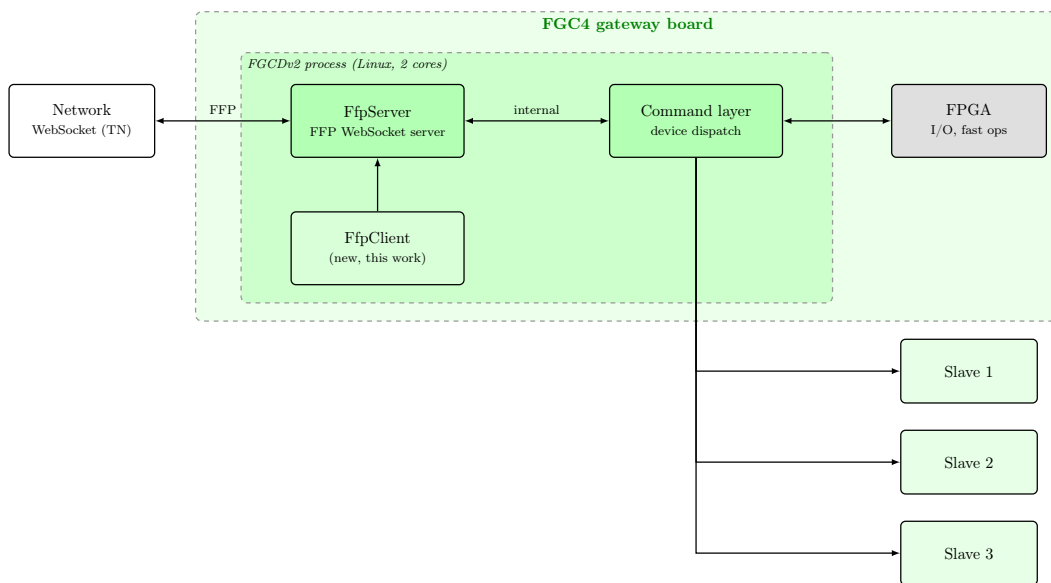


Figure 4.19: Deployment topology of FGCDv2 on the FGC4 gateway board. FGCDv2 runs as a Linux process on two of the four CPU cores and exposes an FFP WebSocket server (**FfpServer**). The **FfpClient** class (introduced in this work, section 4.12.1) provides a C++ client for this same interface. The command layer dispatches requests to the FPGA, which handles I/O and fast operations and communicates with up to three slave boards over the fieldbus.

4.12.1. FFP client implementation (FGCD-122)

The `FfpClient` class was written from scratch as the C++ counterpart of the Python `AsyncConnection` with `FFPProtocol`: it provides a typed, thread-safe interface for connecting to an FFP server over WebSocket and sending protocol requests. The class is defined in `source/modules/gateway/protocols/ffp/client/` and inherits from `Component`, the FGCDv2 lifecycle base class, and from `std::enable_shared_from_this<FfpClient>`, which enables the class to produce a valid `shared_ptr` to itself when registering callbacks with the underlying `WebsocketClient`. The constructor takes a `Component&` parent (for integration into the FGCDv2 component hierarchy) and a `WebsocketClientPtr`, transferring ownership of the WebSocket connection to `FfpClient`. Figure 4.20 shows the class structure.

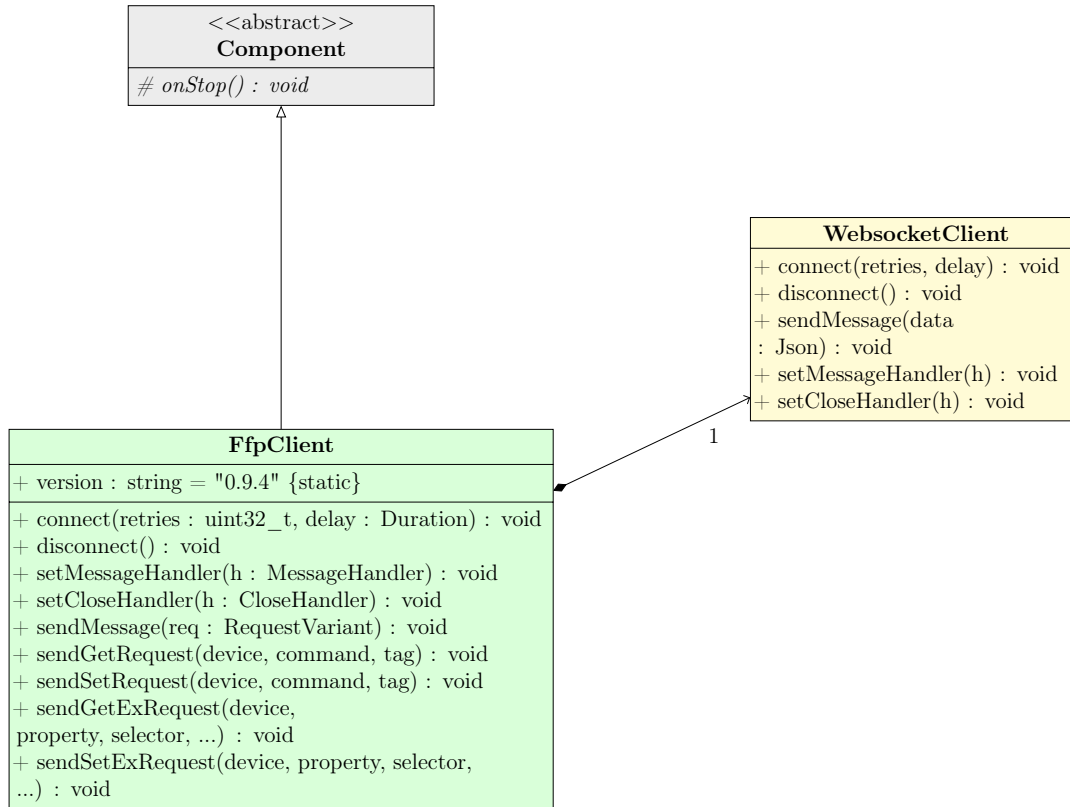


Figure 4.20: Class structure of `FfpClient`. The class inherits from `Component` for FGCDv2 lifecycle management and owns a `WebsocketClient` instance (composition). Green fill indicates the class introduced in this work; grey fill indicates an abstract base class from the FGCDv2 framework. `WebsocketClient` is a pre-existing FGCDv2 class.

The public interface is divided into three groups. Connection management provides `connect()`, which takes a retry count and a minimum delay between retries, and `disconnect()`, which closes the WebSocket cleanly; both delegate directly to the owned

`WebsocketClient`. Lifecycle termination is handled by the protected `onStop()` override, which calls `disconnect()` when the FGCDv2 framework stops the component. Event handler registration provides `setMessageHandler()` and `setCloseHandler()`, both of which are thread-safe: access to the stored handlers is protected by a `std::mutex` guarded with `std::scoped_lock`, so handlers can be registered or replaced from any thread without data races.

The request interface provides four typed shorthand methods and one generic method. The shorthand methods `sendGetRequest()` and `sendSetRequest()` accept a device name, a compact command string, and an optional tag; `sendGetExRequest()` and `sendSetExRequest()` additionally accept a selector, a transaction ID, a filter, and (for the SET variant) a value, matching the explicit field layout of the GETEX and SETEX request types. Each shorthand method constructs the corresponding request struct, populates its fields, and delegates to the generic `sendMessage(const RequestVariant&)`, which uses `std::visit` to serialize the variant to JSON and forward it to the underlying WebSocket. This design keeps the typed methods free of serialization logic while also allowing callers to send any request type including INIT, CLEAR, SUB, UNSUB, and TERM directly via `sendMessage()` without additional convenience wrappers.

The protocol version ("0.9.4") is declared as a `static constexpr` member of both `FfpClient` and `FfpServer`. The client sends its version string in the INIT request and the server echoes its own in the INIT response, making a version mismatch detectable at handshake time. Incoming responses are deserialized inside the private `onMessage()` method, invoked by `WebsocketClient` on the network thread: `onMessage()` calls `json.get<ResponseVariant>()` to produce a typed `ResponseVariant`, then acquires the handler mutex and dispatches to `mMessageHandler` if one is registered. Figure 4.21 shows the full message flow for a GET request.

4.12.2. Error handling standardisation (FGCD-211)

The fgc-next codebase previously relied on `ez::Result` and `ez::Failure` from *Easify*, an internally developed library of generic helper components. Over time, *Easify* became difficult to maintain, extend, and test, placing an increasing burden on the team. The agreed strategy to address this is to remove *Easify* incrementally, one component at a time, replacing each with either project-local code, which is simpler to maintain and test because it does not need to be as generic, or with established external libraries, while avoiding the introduction of further large general-purpose dependencies such as Boost. As part of this work, `ez::Result` and `ez::Failure` were replaced with project-local equivalents:

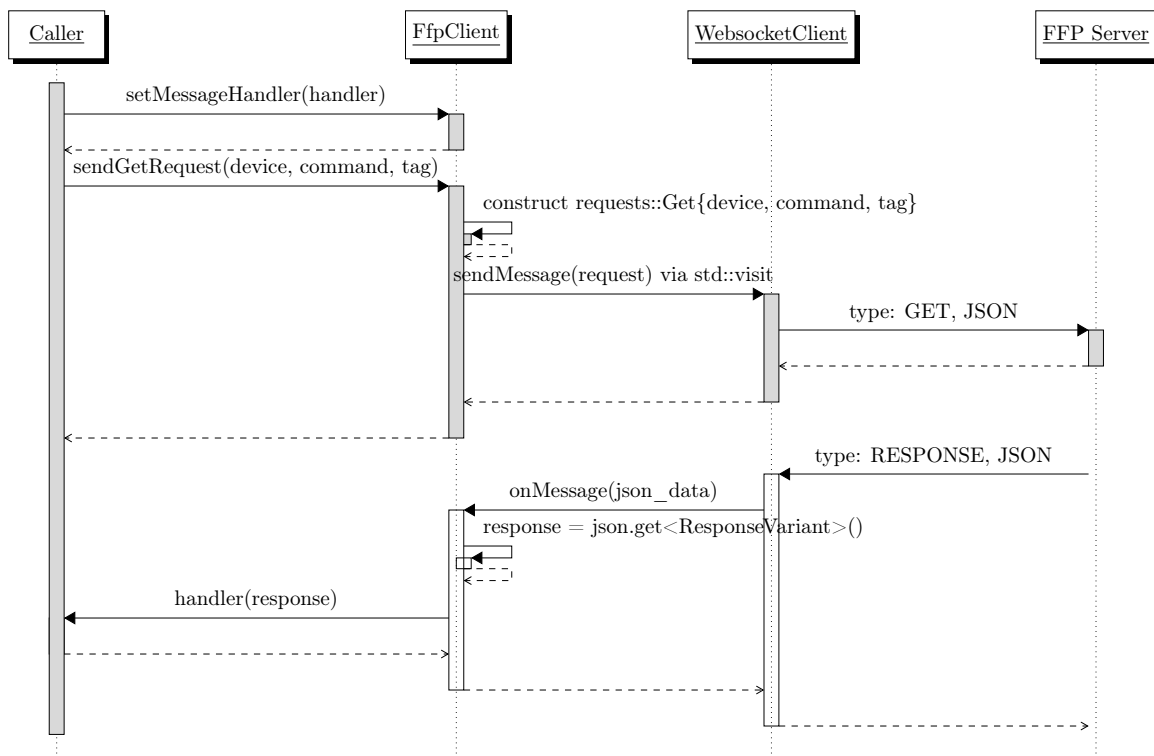


Figure 4.21: Message flow for a GET request through `FfpClient`. The caller registers a message handler, then calls `sendGetRequest()`. `FfpClient` constructs a `requests::Get` struct and forwards it to `WebsocketClient` via `sendMessage()` and `std::visit`. When the server's JSON response arrives, `WebsocketClient` calls `onMessage()`, which deserializes the payload into a `ResponseVariant` and dispatches it to the registered handler. The response path is asynchronous: `sendGetRequest()` returns as soon as the message is sent, and the handler is invoked later on the network thread.

`fgc::Result<T, E>` and `fgc::Failure<E>`, defined in `source/modules/common/types/inc/`.

`fgc::Result<T, E>` is a thin alias over `std::expected<T, E>` from C++23. `fgc::Failure<E>` is a lightweight wrapper around an error value that implicitly converts to `std::expected<T, E>` for any `T`, eliminating the need to repeat the value type at every error return site. These two types together provide the same ergonomics as the replaced `ez` types while depending only on the C++ standard library. The replacement was applied across 35 files in the repository.

4.12.3. Integer type standardisation (FGCD-212)

As part of the same effort to remove Easify (section 4.12.2), a related refactoring removed `ez::numeric`, a set of type aliases for fixed-width integers (`ez::u8`, `ez::i32`, and so on). All occurrences were replaced with the corresponding standard aliases (`uint8_t`, `int32_t`) from `<cstdint>`. This change affected 141 files and had no semantic impact, but eliminates a dependency on Easify and aligns the codebase with idiomatic C++.

4.12.4. FFP protocol interface cleanup (FGCD-221)

The FFP protocol module in `fgc-next` contains both a server implementation (`FfpServer`) and the newly added client (`FfpClient`), together with shared types for requests, responses, and protocol state. The shared headers define the protocol vocabulary used by both classes. `common.hpp` declares the `ffp::MessageType` enum, which enumerates all 13 protocol message kinds (INIT, GET, GETEX, SET, SETEX, SUB, SUBEX, UNSUB, CLEAR, TERM, RESPONSE, NOTIFICATION, STATUS), and the `ffp::STATUS` enum modelling HTTP-like status codes used in INIT and error responses (200 for success; 400, 401, 403, and 404 for error conditions). `requests.hpp` and `responses.hpp` each define one plain struct per message type, aggregate them into a `std::variant<>` alias (`RequestVariant` and `ResponseVariant`), and derive per-struct JSON serialization and deserialization via the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro; type-discriminated deserialization is handled by a custom `nlohmann::adl_serializer` specialization on each variant type that inspects the `"type"` field before selecting the target alternative. The server-side `commandParser.hpp` header declares `CommandParser`, which parses the compact GET and SET command strings into a `CommandFields` struct (property, selector, transaction ID, filter, and remainder) using sequential regex passes, since the optional fields may appear in any order within the command string.

As part of FGCD-221, the public header files for these components were reviewed and

updated: undocumented members received Doxygen comments, inconsistent naming conventions were corrected, and redundant or misleading comments were removed. Seven header files were affected. The `FfpServer` class itself was written by other team members; this contribution covers only the cleanup of its public interface documentation.

At the time of writing, `FfpServer` handles five request types: `INIT`, `GET`, `GETEX`, `SET`, and `SETEX`. Support for the remaining client-to-server types (`SUB`, `SUBEX`, `UNSUB`, `CLEAR`, and `TERM`) requires further implementation work and has not yet been completed. This is consistent with the absence of subscription tests in the integration test suite (section 5.3.6).

4.13. Risks and trade-offs

The primary risk identified during development is the dependency on real FGC4 hardware for integration testing. Since FGC4 is still in active development, the test bench configuration may change, and some tests may need to be re-run as the firmware and FFP server evolve. To mitigate this, the integration test suite is designed to be re-runnable with minimal configuration changes.

A design trade-off was made in the `AutoConnection` layer: by abstracting over protocol and channel selection, the high-level API does not expose all FFP-specific request types (such as `TERM` and cycle-selector-aware `SUBEX` options). Users who require these features must use the lower-level `AsyncConnection` API directly. This trade-off was accepted in favour of a simpler, safer default interface for the majority of use cases.

The following chapter presents the validation strategy used to verify the correctness, reliability, and robustness of the software stack described above, and reports the results obtained against real FGC4 and FGC3 hardware.

5 | Validation and results

5.1. Validation goals

The validation effort has two main goals. The first is to verify that the new `pyfgc 3.0.0` ecosystem behaves correctly in isolation: that each module produces the expected outputs for valid inputs and raises the expected exceptions for invalid ones. The second is to verify that the full communication stack works end-to-end: that a Python client using `pyfgc 3.0.0` can establish a connection to an FGC4 device via FFP, read and write device properties, and receive subscription notifications, with results that match those obtained via the existing NCRP-based toolchain on equivalent FGC3 devices.

Validation is therefore structured into two levels: *unit testing*, which targets individual classes and functions in isolation, and *integration testing*, which targets the assembled stack running against real hardware.

5.2. Validation strategy

5.2.1. Validation environment

Unit tests run without any hardware dependency and are executed automatically on every commit via the GitLab CI/CD pipeline. Each package provides its own test suite under a `tests/` subdirectory, using the `pytest` framework.

Integration tests require a connection to an FGC gateway and a valid RBAC token. They are run manually against the FGC4 test bench available at CERN and, where applicable, against FGC3 devices to verify backward compatibility. Hardware-dependent tests are physically separated from the unit test suite: they reside in a dedicated `integration_tests/` directory, which is excluded from the test paths discovered by the CI pipeline, rather than being mixed into the unit test suite and disabled with skip markers. As a result, integration tests are not part of the CI pipeline at all: no FGC4 device with the FFP server is permanently available for continuous testing, and no dedicated FGC3 or

FGC2 device has been set aside for this purpose either. Integration tests are therefore run manually whenever suitable hardware is available. Chapter 6 discusses plans to address this limitation.

5.2.2. Testing strategy

The testing approach differs by package and component. Five strategies are employed across the ecosystem.

Data-driven protocol tests The core of the pyfgc unit test suite is a set of data-driven tests for the protocol layer. Each protocol (FFP, NCRP, SCRP, CMW) is covered by three dedicated test files targeting, respectively, the message type (`test_ffp_message.py` and equivalents), the request generator, and the response parser. Tests load pre-captured protocol messages from JSON fixture files stored under `tests/data/` and assert that the parsed output matches the expected structured representation. This approach decouples test logic from message construction, makes the expected behaviour explicit and reviewable, and ensures that the parser is validated against real protocol messages rather than synthetic inputs.

Mock server and device emulation To test the channel and connection layers without real hardware, the test suite provides a set of in-process mock servers and protocol emulators. At the lowest level, `MockWebSocketsServer` and `MockTCPIPServer` are fully functional network servers that bind to a temporary localhost port and handle connections over the standard socket interface. Tests connect to these servers using the same `AsyncWebSocketsChannel` and `AsyncTCPIPChannel` classes that would be used against real hardware, exercising the full send/receive path including framing, encoding, and error handling.

On top of these transport servers, `MockFFPProtocol` and `MockNCRPProtocol` implement partial but fully functional server-side versions of the respective protocols. `MockFFPProtocol` handles INIT, GET, GETEX, SET, and SETEX request types, enforces the mandatory INIT handshake, validates device and property names, and optionally validates RBAC token credentials. `MockNCRPProtocol` handles GET and SET commands with the full NCRP framing and TAG-based response correlation. Both protocol emulators share a `TestProperties` instance, which is an in-memory property store containing a fixed set of typed properties, listed in table 5.1. These properties are not specific to the mock servers: they are defined on `FGCD_BASE`, the common base class inherited throughout the FGC class hierarchy [25], and are present on most real devices for the sole purpose of

testing and validation, exposing one read/write property per supported scalar type with no operational meaning. Range validators on each property allow SET commands to be exercised with both valid and out-of-range values.

Table 5.1: Typed properties exposed by `TestProperties` in the mock protocol servers, with their initial value and accepted range.

Property	Type	Initial value	Accepted range
TEST.CHAR	string	" "	length ≤ 64
TEST.FLOAT	float	0.0	$-10^{30} < x < 10^{30}$
TEST.INT16S	int16	0	-32768 to 32767
TEST.INT16U	uint16	0	0 to 65535
TEST.INT32S	int32	0	-2147483648 to 2147483647
TEST.INT32U	uint32	0	0 to 4294967295
TEST.INT8S	int8	0	-128 to 127
TEST.INT8U	uint8	0	0 to 255

FGC4MockDevice and FGC3MockDevice compose these building blocks into complete device emulators: FGC4MockDevice wires MockWebSocketsServer with MockFFPProtocol to simulate an FGC4 device over WebSockets; FGC3MockDevice wires MockTCPIPServer with MockNCRPProtocol to simulate an FGC3 device over TCP/IP. Tests that target the connection and AutoConnection layers instantiate these mock devices, start them in the background, then exercise the full client API against them. Because the tests use real socket connections to real servers, the entire client-side stack from the Python API down to network I/O is exercised, without having to patch any of the components.

SyncWrapper API surface enforcement A distinct set of test files (`test_sync_async_channel_methods.py`, `test_sync_async_connection_methods.py`, and three equivalent files for factories and AutoConnection) uses Python’s `inspect` module to enforce the architectural invariant that the synchronous and asynchronous APIs are always in parity. For each pair of async and sync classes, the tests verify that every public method of the async class is also present in the sync wrapper and that the sync method is not a coroutine. This provides automated enforcement of the design contract described in section 4.8: if a method is added to the async API without a corresponding entry in the sync wrapper, the test fails.

Fixture-based name and token tests `pyfgc_name` tests run against static mock name and group data files bundled with the test suite, making them independent of the live operational files at CERN. Similarly, `pyfgc_rbac` tests for the RBACToken inter-

face (expiry checking, role querying, serialisation) use pre-captured token bytes stored in `tests/data/rbac_token/`. Token generation tests for Kerberos and location-based authentication run against a mock RBAC server and therefore require no live infrastructure.

HTTP layer tests All 89 tests in `ccs-fastapi-utils` use FastAPI’s `TestClient` to exercise the full HTTP request-to-response path, including exception handlers, middleware, and RFC 9457 problem details response formatting, without mocking any HTTP component. This ensures that the RBAC authentication middleware and error handling logic are validated end-to-end rather than in isolation.

5.2.3. Acceptance criteria

For unit tests, the acceptance criterion is that all tests pass and that branch coverage across the test suite is above an agreed threshold (75%). For integration tests, the criteria are: successful property reads and writes returning values consistent with the device state; subscription notifications arriving within the expected timing; RBAC authentication succeeding with a valid token and failing with an invalid or expired token; and backward-compatible behaviour on FGC3 devices verified by comparing `pyfgc 3.0.0` results against those from `pyfgc 2.x` for the same property accesses.

5.3. Validation results

5.3.1. Unit tests

The unit test suite across the six packages comprises 946 tests in total, run under Python 3.14.2 against the `sprint` branch. Table 5.2 reports the distribution and branch coverage per package.

Table 5.2: Unit test counts and branch coverage by package (sprint branch, Python 3.14).

Package	Tests	Branch coverage
<code>pyfgc</code>	664	97%
<code>pyfgc_name</code>	75	99%
<code>pyfgc_rbac</code>	78	100%
<code>pyfgc_statussrv</code>	8	83%
<code>pyfgc_decoders</code>	18	100%
<code>ccs-fastapi-utils</code>	103	85%
Total	946	—

All 946 tests pass without hardware access. The SyncWrapper API surface tests (section 5.2.2) run as part of this test suite and confirm that all async methods in channels, connections, factories, and AutoConnection classes have correctly wrapped synchronous counterparts. All six packages exceed the agreed 75% branch coverage threshold (section 5.2.3).

The uncovered branches in `pyfgc_statussrv` (83%) correspond to the network I/O paths of `StatusServerClient`, which requires a live status server and is therefore not exercised by unit tests. The uncovered branches in `ccs-fastapi-utils` (85%) correspond to the RBAC token renewal flow in `token.py` and the OAuth2/SAML authentication paths in `flows.py`, both of which require a live RBAC server. These gaps are expected, do not indicate defects in the tested logic, and the cost of mocking the external dependencies to cover them would outweigh the benefit.

By contrast, the predecessor ecosystem had virtually no meaningful automated coverage, as detailed in section 3.1.12. In the new ecosystem, no test file is commented out, and no test function is left as a stub; where hardware or live infrastructure access is required, tests are explicitly separated into a labelled integration test stage rather than left as disabled stubs within the unit suite. The entire unit test suite runs in the GitLab CI pipeline on every commit without requiring device access, while integration tests against live FGC3 and FGC4 devices form a separate, explicitly labelled test stage rather than being left as informal manual checks or disabled stubs. The 946-test suite, all passing without hardware, represents a substantial improvement in verifiability and regression protection.

5.3.2. Static analysis and code quality

Static type checking was performed manually on all new packages using both `mypy` and `pyright`, the latter configured in strict mode. All new code passes without errors or warnings. Integration into the CI pipeline was not undertaken at this stage: the `fgc` monorepo contains pre-existing packages without complete type annotations, and adding a type-checking job to the shared pipeline would have required retrofitting those packages. This remains an open item for future work.

Code style and lint checks using `ruff` and `pylint` are configured for all new Python packages and run as part of the development workflow. All new code passes both tools without errors or warnings.

For the `fgc-web` repository, a SonarQube analysis job runs on every merge request and evaluates combined coverage reports from all applications against a shared quality gate, as described in section 4.11.3. The quality gate is configured with a threshold of zero errors;

all issues reported by SonarQube for the modified applications were resolved before the changes were merged.

All changes were submitted as merge requests and underwent peer review by at least one other member of the SY-EPC-CCS section before being accepted, in accordance with the team's standard development process (section 4.11.1).

5.3.3. Integration tests

Integration tests were executed against two live devices at the SY-EPC-CCS test bench: an FGC4 device accessed via FFP (CFE-866-RDEV3, WebSocket port 6769) and an FGC3 device accessed via NCRP (RFNA.866.01.ETH1, TCP port 1905). RBAC credentials were obtained via Kerberos authentication (`kinit`) prior to each test run.

FGC4 via FFP The following tests were executed against CFE-866-RDEV3 and all passed:

- **Connectivity:** a TCP socket connection to the WebSocket port was established successfully.
- **INIT handshake:** the FFP session was opened with a valid RBAC token; the server returned HTTP 200. Both client and server advertised protocol version "0.9.4", confirming protocol conformance between the Python client and the FGCDv2 firmware.
- **Property GET and SET (scalar):** all eight property types exposed by the test device (`test.char`, `test.float`, `test.int16s`, `test.int16u`, `test.int32s`, `test.int32u`, `test.int8s`, `test.int8u`) were written and read back via both the compact (GET/SET) and extended (GETEX/SETEX) request forms. Read-back values matched the written values for all types.
- **Command tag correlation:** the TAG field was set on each request and verified to be echoed correctly in every response, confirming that response-to-request matching works when multiple requests are in flight.
- **Random round-trip:** 100 iterations of SET followed by GET with randomly generated values were executed for `test.int32s` and `test.char`; all read-back values matched the written values exactly.
- **RBAC:** a valid Kerberos-derived token was accepted by the server. An invalid token (random bytes) was correctly rejected.

The FFP message types SUB, SUBEX, UNSUB, and TERM were not exercised against

CFE-866-RDEV3: server-side push notification support was not yet available on the FGC4 test bench at the time of testing, and TERM was not independently exercised beyond the implicit session teardown. These message types are covered at the unit-test level by the mock FFP server, which handles SUBEX acknowledgement, notification dispatch, and UNSUB. They will be added to the integration test suite once the firmware supports them.

Device name resolution Device name resolution was exercised as part of every integration test: `AsyncAutoConnectionFactory.auto_create_connection()` resolved the device name to a gateway address and class identifier via `pyfgc_name` before establishing the connection, fetching the live name files from CERN’s internal name server over HTTP. All resolutions succeeded, confirming that the full name-resolution path operates correctly against real infrastructure. The parsing and validation logic of `pyfgc_name` is additionally covered by 75 unit tests at 99% branch coverage, using static mock name files bundled with the test suite.

The C++ `FfpClient` (FGCD-122) was first validated manually by running it against an FFP server instance. All FGC4 integration tests subsequently exercised the FFP stack end-to-end through the FGCDv2 firmware on the test bench, further confirming the interoperability of the C++ implementation with the Python client library.

FGC3 via NCRP The following tests were executed against `RFNA.866.01.ETH1` and all passed:

- **Connectivity:** a TCP socket connection to port 1905 was established successfully.
- **RBAC token login:** the RBAC token was set via the `CLIENT.TOKEN` property using both the packed binary encoding and the object-level API; both were accepted and the server response confirmed success.
- **Property GET (scalar):** all eight scalar types were read successfully; returned values matched the expected device state.
- **Property SET and GET round-trip (scalar and vector):** scalar and array-valued properties were written and read back; all values matched.
- **Out-of-range SET:** SET commands with values outside each property’s valid range were rejected by the device with the expected error response.
- **PPM selector:** a GET with a user selector (`PSB.USER.EAST3`) was issued for a timing-sensitive property (`MEAS.DIRECT.I`) and returned a valid response, confirming that NCRP cycle-specific access works correctly.

Status server `pyfgc_statussrv` 2.0.0 was tested against a live FGC status server instance. The decoded status records were validated manually against known device states and cross-checked by comparing the output of `pyfgc_statussrv` with the device state indicators displayed by both FGC API and FGC Commander, both of which consume the status server through the same library. Agreement across all three views confirmed the correctness of the binary decoding and the `StatusServerClient` interface.

CMW/RDA subscription The CMW/RDA subscription support added in EPCTOOLS-292 was tested against a live RDA server instance running on an FGC3. The `AsyncAutoConnection.subscribe()` and `AsyncAutoConnection.unsubscribe()` methods were exercised with a cycle selector, confirming that the `RDAChannelListener` correctly dispatches incoming notifications and that the subscription lifecycle operates as described in section 4.6.

Application regression PowerSpy 3.0.2 and FGC Commander, both migrated to `pyfgc` 3.0.0 as part of this work, were smoke-tested after migration and subsequently validated over an extended period of manual use. A parallel comparison was also conducted: the old and new versions of PowerSpy, FGC Commander, and the FGC API were run side by side against the same devices, and their outputs were compared to confirm consistency. No discrepancies were observed, confirming that the migration introduced no regressions in application behaviour.

5.3.4. Performance

End-to-end latency of FFP requests depends primarily on the server-side processing time in the FGCDv2 firmware rather than on the Python client library, and varies with device load and network conditions. A dedicated performance characterisation is therefore outside the scope of this work and is left as future work once the FGC4 hardware platform is in a more stable state.

5.3.5. Compatibility

Backward compatibility with FGC3 devices was verified against `RFNA.866.01.ETH1` using `pyfgc` 3.0.0 with the NCRP channel. The RBAC token encoding used by the new `RBACToken.packed` property was verified to produce bit-for-bit identical output to the manual packing previously required by `pyfgc` 2.x, confirming that the authentication handshake is compatible. Scalar and vector property GET and SET operations returned correct values across all tested property types, and out-of-range writes were rejected with

the expected error responses. No regressions were observed relative to the behaviour expected from the FGC3 NCRP specification.

5.3.6. Reliability and robustness

Four scenarios targeting failure modes and edge conditions were evaluated against the FGC4 test bench.

Connection loss and reconnection FFP sessions are stateful: the server maintains per-connection state established during the INIT handshake. When a WebSocket connection is dropped, `pyfgc` raises an exception at the point of the failed operation; the caller is responsible for establishing a new connection and re-issuing INIT before resuming normal operation. In the FGC API service, connection loss is handled transparently: the request handler creates a new `AutoConnection` for each incoming request via the factory, so INIT is re-issued automatically as part of connection creation and no special recovery logic is required at the application level.

Malformed and out-of-range requests Request validation in FFP is performed server-side. Requests referencing a non-existent property return `UNKNOWN_SYMBOL` (code 17); requests naming a device unknown to the server return `UNKNOWN_DEVICE` (code 38); SET commands with values outside the property’s valid range return `OUT_OF_LIMITS` (code 28); syntactically malformed commands return `SYNTAX_ERROR` (code 14) or `BAD_PARAMETER` (code 23). In all cases the server returns a well-formed FFP response carrying the appropriate `FFPCode` value without terminating the connection. The client library propagates these codes through the `AutoResponse.error_code` field without raising exceptions, in accordance with the explicit error-handling design described in section 4.6.

Concurrent requests One hundred concurrent requests were issued against CFE-866-RDEV3 using `AsyncAutoConnection`. Each of the 100 asyncio tasks acquired an `asyncio.Semaphore` before establishing its own connection and issuing GET and SET requests on a scalar test property, using a per-task command tag to verify response correlation. Under the FFP protocol, `AsyncAutoConnection.get()` and `AsyncAutoConnection.set()` internally issue GETEX and SETEX request types respectively, so all four compact and extended request variants were exercised. All requests completed successfully: no errors, lost messages, or incorrect correlations were observed. The structure of the test is shown in listing 5.1.


```

1  import asyncio
2  from pyfgc.autoconnection.factory import AsyncAutoConnectionFactory
3
4  DEVICE = "CFE-866-RDEV3"
5  CONCURRENCY = 100
6
7  async def stress_task(sem: asyncio.Semaphore, idx: int) -> None:
8      async with sem:
9          conn = await AsyncAutoConnectionFactory.
10             auto_create_connection(DEVICE)
11      async with conn:
12          tag = str(idx)
13          r = await conn.set("TEST.FLOAT", 1.0, command_tag=tag)
14          assert r.error_code == 0
15          assert r.command_tag == tag
16          r = await conn.get("TEST.FLOAT", command_tag=tag)
17          assert r.error_code == 0
18          assert r.command_tag == tag
19          assert r.value == 1.0
20
21  async def run() -> None:
22      sem = asyncio.Semaphore(CONCURRENCY)
23      await asyncio.gather(*(stress_task(sem, i) for i in range(
24          CONCURRENCY)))
25
26  asyncio.run(run())

```

Listing 5.1: Structure of the concurrent-request stress test against CFE-866-RDEV3.

The FGC API HTTP layer was stress-tested separately using the Locust load testing framework against the GET, SET, and device listing endpoints. A swarm of 1000 simulated users issued requests concurrently against a running FGC API instance; all requests succeeded. Response latency is not reported, as it depends primarily on the load of the host running FGC API and on the device under test rather than on the client library itself.

Long-running sessions A continuous session of 24 hours was maintained against the FGC4 test bench, issuing periodic GET requests, from both the `pyfgc` library directly and via the FGC API service. No connection drops, memory growth, or degradation in response correctness were observed over the full duration. RBAC tokens were renewed transparently by the `AutoConnection` token renewal mechanism over the course of the session, confirming that the automatic renewal logic described in section 4.6 operates correctly without caller intervention over extended periods.

A separate 48 hour session was conducted against `CFE-866-RDEV3`, again from both `pyfgc` directly and the FGC API service, specifically to validate the RBAC token lifecycle under sustained load. The test exercised token acquisition, expiry detection, and automatic renewal through multiple full token lifetime cycles, verifying that the library maintains authenticated access continuously without requiring caller intervention. No authentication failures or stale-token errors were observed over the full duration.

5.4. Validation summary

Table 5.3 summarises all validation activities carried out in this work and their outcomes. Activities are grouped into three categories: static analysis and code quality, which require no hardware and run automatically in CI; integration testing, which exercises the full communication stack against real devices; and robustness testing, which verifies correct behaviour under sustained or concurrent load.

Taken together, these results demonstrate that the software stack is functionally correct, structurally sound, and operationally reliable. The unit test suite and static analysis tools provide a continuous quality baseline that runs automatically on every code change. Integration tests confirm end-to-end correctness against both FGC4 and FGC3 hardware, and backward compatibility with the existing installed base. Robustness tests show that the stack sustains high concurrency and long sessions without degradation, which is a prerequisite for deployment in an operational accelerator environment where connections are expected to remain open across entire accelerator cycles.

The following chapter summarises the overall results of this work, reflects on the contributions in the context of the FGC4 deployment roadmap, and identifies directions for future development.

Table 5.3: Summary of validation activities and results.

Activity	Method	Result
<i>Static analysis and code quality</i>		
Unit tests (946 total)	Automated, GitLab CI	All pass
Branch coverage	<code>pytest-cov</code> , per package	83%–100%
Static type checking	<code>mypy</code> + <code>pyright</code> (strict)	Zero errors
Linting	<code>ruff</code> + <code>pylint</code>	Zero errors
SonarQube quality gate	Automated, CI (<code>fgc-web</code>)	Passed
Peer code review	MR review, SY-EPC-CCS team	Approved
<i>Integration testing (real hardware)</i>		
FGC4 integration (FFP/WS)	Live device <code>CFE-866-RDEV3</code>	All passed
FGC3 integration (NCRP)	Live device <code>RFNA.866.01.ETH1</code>	All passed
Device name resolution	Live name server (HTTP)	All resolved
Status server	Live server, cross-validated	Correct
CMW/RDA subscription	Live RDA server, manual	Correct
FGC API, PowerSpy, Commander	Parallel comparison, old vs. new	No regressions
Backward compatibility	<code>pyfgc</code> 3.0.0 vs. 2.x on FGC3	No regressions
<i>Robustness testing</i>		
Concurrent requests	100 asyncio tasks (<code>pyfgc</code>); 1000 simulated users (FGC API)	All succeeded
Long-running session (24 h)	<code>pyfgc</code> + FGC API vs. <code>CFE-866-RDEV3</code>	No drops or errors
RBAC token lifecycle (48 h)	<code>pyfgc</code> + FGC API vs. <code>CFE-866-RDEV3</code>	No auth failures

6 | Conclusions and future developments

6.1. Summary of the results

This thesis presented the design and implementation of the software communication stack required to integrate FGC4 devices into CERN’s power converter control infrastructure. The work addressed two interconnected challenges: the functional absence of FGC4 support in the existing `pyfgc` ecosystem, and a set of structural software quality deficiencies that made the ecosystem difficult to maintain and extend.

The primary contribution is `pyfgc` 3.0.0, a complete rewrite of the Python library for FGC device communication. The new library introduces WebSocket transport and FFP protocol support, enabling communication with FGC4 for the first time. Its architecture cleanly separates the transport layer (channels) from the message serialization layer (protocols) through a pair of generic abstractions, making it straightforward to add new transports or protocols in the future. A `SyncWrapper` utility generates all synchronous APIs from their asynchronous counterparts, eliminating the code duplication that afflicted the previous version. The factory pattern is applied consistently at the `Connection` and `AutoConnection` layers, each exposing a paired `AsyncConnectionFactory` / `SyncConnectionFactory` and `AsyncAutoConnectionFactory` / `SyncAutoConnectionFactory`, so that object construction follows the same async-first, sync-wrapped discipline as the rest of the library. Protocols are also named after the wire protocol they implement (`FFPProtocol`, `NCRPProtocol`, `SCRPProtocol`, `CMWProtocol`), replacing the previous `sync/async/serial/rda` naming, which described the transport and execution mode rather than the protocol and did not match the terminology used by the teams operating these services. The high-level `AutoConnection` layer provides a simple, unified interface that automatically resolves device names, selects the appropriate protocol based on device generation, and manages RBAC token lifecycle, resolving the manual orchestration burden identified in section 3.1.11.

The companion packages `pyfgc_name` 3.0.0, `pyfgc_rbac` 3.0.0, and `pyfgc_statussrv` 2.0.0 were rewritten alongside the main library, replacing global mutable state with instance-based designs, introducing typed entity classes, and adopting modern Python type annotations throughout. In particular, `pyfgc_rbac` 3.0.0 replaces the opaque byte-string token with a structured `RBACToken` object that exposes expiry information directly, eliminating both the manual payload decoding and the reliance on an untyped, undocumented dictionary whose field names had to be inferred from the token payload rather than from any published API contract. The `pyfgc_name` 3.0.0 `Names` class replaces the module-level dictionaries with typed `Device`, `SubDevice`, and `Group` objects, giving callers static type guarantees in place of untyped dictionary access. The `pyfgc_rbac` package additionally introduces the structured `RBACToken` class described above, encapsulating token acquisition and renewal behind a typed interface in place of the previous untyped dictionary, while retaining the Kerberos/GSSAPI authentication already supported by the previous version. All packages ship with comprehensive test suites and generated API documentation. Together, these changes resolve the structural deficiencies identified in chapter 3.

The C++ FFP server, running on the FGCDv2 framework on FGC4 hardware, provides the server-side counterpart to the Python client library. Together they constitute a complete, validated communication stack for FGC4.

Validation was carried out at two levels. The unit test suite spans six packages and comprises 946 tests in total, all of which pass without hardware access under Python 3.14.2. Branch coverage is 97% for `pyfgc`, 99% for `pyfgc_name`, 100% for `pyfgc_rbac` and `pyfgc_decoders`, 85% for `ccs-fastapi-utils`, and 83% for `pyfgc_statussrv` (the uncovered branches in the last two require a live RBAC or status server and are expected gaps). Integration tests were executed against two real devices at the SY-EPC-CCS test bench: an FGC4 device accessed via FFP (`CFE-866-RDEV3`) and an FGC3 device accessed via NCRP (`RFNA.866.01.ETH1`). All integration tests passed, confirming correct end-to-end behaviour for both device generations and backward compatibility with the existing FGC3 infrastructure. Robustness tests further confirmed that the stack handles concurrent load correctly: 100 concurrent GET and SET requests were served without message loss or incorrect correlation through `AsyncAutoConnection` directly, and all requests from a swarm of 1000 simulated users succeeded through the FGC API HTTP layer. Continuous 24-hour sessions maintained against the FGC4 test bench produced no connection drops or memory growth, from both the `pyfgc` library and the FGC API service.

6.2. Future developments

Several directions are identified for future work. First, as FGC4 devices are deployed in the LHC and, in the longer term, in the Future Circular Collider (FCC) [5], the integration test suite should be extended to cover a broader range of device classes and operational scenarios. At present, integration tests are run manually whenever suitable hardware becomes available, since no FGC4 device with the FFP server is permanently reserved for this purpose, nor is any FGC3 or FGC2 device dedicated to it. A dedicated set of FGC4 devices, together with representative FGC3 and FGC2 devices, is planned to be acquired exclusively for automated nightly end-to-end testing, providing a stable, controlled hardware environment for systematic regression testing as the firmware evolves; this hardware is not yet available at the time of writing. The `AutoConnection` layer currently exposes only the common subset of FFP and NCRP request types; future work could expose FFP-specific features such as cycle-selector-aware subscriptions to applications that require them.

Second, the `pyfgc_statussrv` package currently relies on NCRP for status server communication. As FGC4 devices expose their status via FFP, this package should be extended to support FFP-based status retrieval.

Third, the FGC4 deployment may eventually motivate a higher-level abstraction that allows control applications to target a heterogeneous fleet of FGC3 and FGC4 devices transparently, without needing to know which protocol is in use at any given site. The `AutoConnection` layer already provides the foundation for such an abstraction.

Finally, as the `pyfgc` ecosystem stabilises, the opportunity exists to extend the same Channel/Protocol/Connection architecture to other CERN control system libraries that face similar protocol migration challenges. In particular, discussions within the team have considered a future migration toward Tango, a widely used open-source control framework. The layered architecture introduced in `pyfgc` 3.0.0 is well-suited to such a migration: adding Tango transport support would require implementing a new `Channel` subclass without modifying any existing protocol or connection logic, in the same way that the `AsyncWebSocketsChannel` was introduced alongside the existing TCP/IP channel to enable FFP support.

Bibliography

- [1] ATLAS Collaboration. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Physics Letters B*, 716: 1–29, 2012. doi: 10.1016/j.physletb.2012.08.020.
- [2] F. Boattini, J.-P. Burnet, and G. Skawinski. POPS: The 60MW power converter for the PS accelerator: Control strategy and performances. In *2015 17th European Conference on Power Electronics and Applications (EPE'15 ECCE-Europe)*, 2015. doi: 10.1109/EPE.2015.7309267.
- [3] CERN. Accelerator report: Target reached, now for the compressor, 2024. URL <https://home.cern/accelerator-report-target-reached-compressor/>.
- [4] CERN. New schedule for CERN’s accelerators, 2025. URL <https://home.cern/new-schedule-cerns-accelerators/>.
- [5] CERN. Future circular collider, 2026. URL <https://home.cern/science/accelerators/future-circular-collider/>.
- [6] CERN Accelerator Controls. ACC-Py: Accelerator python documentation. <https://acc-py.docs.cern.ch/>, 2025. Accessed: 2026-06-17.
- [7] CERN Accelerator Controls. ACC-Py GitLab CI templates. CERN internal GitLab: [acc-co/devops/python/acc-py-gitlab-ci-templates](https://gitlab.cern.ch/acc-co/devops/python/acc-py-gitlab-ci-templates), 2025. Available at <https://gitlab.cern.ch/acc-co/devops/python/acc-py-gitlab-ci-templates>.
- [8] CERN BE-CO. FESA: Front-end software architecture, 2026. URL <https://wikis.cern.ch/display/FESA/>.
- [9] CERN Bulletin. POPS: A new beat for the PS, 2011. URL <https://cds.cern.ch/journal/CERNBulletin/2011/11/News%20Articles/1334847>.
- [10] CERN SY-EPC-CCS. Converter control, 2026. URL <https://epc-ccs.web.cern.ch/content/converter-control>.
- [11] CMS Collaboration. Observation of a new boson at a mass of 125 GeV with the CMS

- experiment at the LHC. *Physics Letters B*, 716:30–61, 2012. doi: 10.1016/j.physletb.2012.08.021.
- [12] CMW - Controls Middleware. RBAC. Technical Report v19, CERN, May 2026. Internal specification. Available at CMW documentation.
- [13] CMW - Controls Middleware. RBAC user guide. Technical Report v47, CERN, May 2026. Internal specification. Available at CMW documentation.
- [14] G. Daniluk, A. Arias Vázquez, A. Pinho, B. Guncic, M. Lettrich, P. Peronnard, and V. Amoiridis. Distributed I/O tier: from concept to operational readiness – a modular platform for custom radiation-tolerant and radiation-free electronics. In *Proceedings of the 20th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2025)*, Chicago, USA, 2025. WEBR006, <https://proceedings.jacow.org/icalepcs2025/pdf/WEBR006.pdf>.
- [15] A. Dworak et al. The new CERN controls middleware. *Journal of Physics: Conference Series*, 396:012017, 2012. doi: 10.1088/1742-6596/396/1/012017.
- [16] L. Evans and P. Bryant. LHC machine. *Journal of Instrumentation*, 3:S08001, 2008. doi: 10.1088/1748-0221/3/08/S08001.
- [17] K. Kahle. Static Var compensation for the SPS electrical network. Technical Report CERN-ST-2000-040, CERN, Feb. 2000. Presented at the 3rd ST Workshop, Chamonix, France, January 25–28, 2000.
- [18] K. Kahle, F. R. Blázquez, and C.-M. Genton. The design and performance of static Var compensators for particle accelerators. In *2015 17th European Conference on Power Electronics and Applications (EPE'15 ECCE-Europe)*, number CERN-ACC-2015-0104, 2015.
- [19] C. Kishimoto and T. Cass. CERN technical network upgrade. Technical report, CERN IT/CS, 2018.
- [20] H. Milcent, X. Genillon, M. Gonzalez-Berges, and A. Voitier. Supervision application for the new power supply of the CERN PS (POPS). In *Proceedings of the 13th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2011)*, pages 756–758, Grenoble, France, 2011. WEPKN025, CERN-ATS-Note-2011-112, <https://cds.cern.ch/record/1400721>.
- [21] SY-EPC-CCS section. CMW/RDA command line tools (RDA2 and RDA3). Technical Report v16, CERN, May 2026. Internal specification. Available at Confluence page 413495201.

- [22] SY-EPC-CCS section. FEC command string format, 2026. URL <https://fgc-docs.web.cern.ch/>. Version 0.3.0.
- [23] SY-EPC-CCS section. SY-EPC-CCS: Converter controls software section, 2026. URL <https://epc-ccs.web.cern.ch/>.
- [24] SY-EPC-CCS section. FFP: FGC front-end protocol, 2026. URL <https://fgc-docs.web.cern.ch/>. Version 0.9.4.
- [25] SY-EPC-CCS section. FGC project site map: Platform and class reference, 2026. URL <https://accwww.cern.ch/proj-fgc/proj-fgc/gendoc/def/SiteMap.htm>. Auto-generated reference of FGC platforms, device classes, and per-class properties.
- [26] SY-EPC-CCS section. FGC command/response protocol. Technical Report v11, CERN, May 2026. Internal specification. Available at Confluence page 414977551.
- [27] SY-EPC-CCS section. **data**: FGC configuration data repository. CERN internal GitLab repository, <https://gitlab.cern.ch/epc/ccs/fgc/data>, 2026. Restricted to CERN members.
- [28] SY-EPC-CCS section. **fgc-web**: FGC web tools repository. CERN internal GitLab repository, <https://gitlab.cern.ch/epc/ccs/tools/fgc-web>, 2026. Restricted to CERN members.
- [29] SY-EPC-CCS section, FGC team. **fgc**: FGC python packages monorepo. CERN internal GitLab repository, <https://gitlab.cern.ch/epc/ccs/fgc/fgc>, 2026. Restricted to CERN members.
- [30] SY-EPC-CCS section, FGC team. **fgc-next**: FGC4 C++ software monorepo. CERN internal GitLab repository, <https://gitlab.cern.ch/epc/ccs/fgc/fgc-next>, 2026. Restricted to CERN members.
- [31] N. Trofimov, V. Baggiolini, S. Jensen, K. Kostro, F. Di Maio, and A. Risso. Remote device access in the new CERN accelerator controls middleware. In *Proceedings of the 8th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALPCS 2001)*, San Jose, California, 2001. THAP003, arXiv:physics/0111166.

List of Figures

1.1	Aerial view of the CERN site near Geneva. The 27 km LHC ring is traced in yellow; the smaller SPS ring (7 km) and the CERN campuses are visible on the top right. (Image credit: CERN)	2
1.2	Artist's rendering of the proposed 91 km FCC tunnel (dashed) overlaid on the existing HL-LHC (white), which would serve as part of its injector chain. (Image credit: CERN)	3
2.1	The CERN accelerator complex. Particles are progressively accelerated through a chain of machines before being injected into the LHC. (Image credit: CERN)	8
2.2	The FGC2 hardware module. (Image credit: CERN)	9
2.3	The FGC3 hardware module. (Image credit: CERN)	10
2.4	The DI/OT hardware kit. The 3U CompactPCI Serial crate (centre) accommodates the System Board ZU (Zynq UltraScale+ MPSoC, non-radiation-tolerant) and the System Board IGL (Igloo2 FPGA, radiation-tolerant, marked with the radiation symbol). Power is supplied by either a COTS PSU or the radiation-tolerant RaToPUS. Application-specific Peripheral Boards (shown left) plug into the backplane. (Image credit: CERN / Open Hardware Repository [14])	11
2.5	The FGC4 prototype board. (Image credit: CERN)	11
2.6	The POPS power converter system. (Image credit: CERN)	13
2.7	The SVC installation affected by the hardware failure. (Image credit: CERN)	15
2.8	CERN network topology. The GPN hosts developer workstations and web-facing applications; the TN hosts FGC controllers and accelerator instrumentation. TN-trusted nodes are dual-homed machines that bridge the two networks while having no internet access themselves.	16
2.9	The FGC Portal, providing a unified entry point to all applications in the FGC control software ecosystem. The selector in the top right corner allows developers to switch between deployment environments (dev, qua, prod).	24

- 2.10 FGC Commander device selection dialogue. The left panel lists all available device groups with regular-expression search; the right panel shows the current selection broken down by device type (in this case, FGC generation). 26
- 2.11 FGC Commander monitoring view, showing real-time device state and measurements for a set of selected FGC devices. The colour-coded status indicators (operational state, voltage/current status, polarity) are obtained via the FGC status server, while the current and voltage reference and measurement values are read directly from the devices. 26
- 2.12 PowerSpy displaying an FFT analysis of current and voltage signals acquired from a power converter. The application supports raw time series, tabular, and frequency-domain views. 27

- 4.1 Package interactions in the `pyfgc` 3.0.0 ecosystem. `pyfgc` is the central library; its `AutoConnection` layer uses `pyfgc_name` for device resolution and `pyfgc_rbac` for token lifecycle management. `pyfgc_statussrv` uses a `pyfgc AsyncConnection` for network communication and delegates binary decoding to `pyfgc_decoders`. Control applications (FGC API, PowerSpy) and engineer scripts consume `pyfgc` through the `AutoConnection` API and acquire RBAC tokens directly from `pyfgc_rbac`. 46
- 4.2 Layered component architecture of `pyfgc` 3.0.0. Each layer communicates with the layer below through a defined interface (shown between rows). `Connection` composes one `Channel` and one `Protocol`; `AutoConnection` owns a `Connection` and adds device name resolution, protocol selection, and RBAC token management. Green fill indicates classes introduced in this version; yellow fill indicates non-standard interface (`RDACHannel`). . . . 47
- 4.3 Channel class hierarchy in `pyfgc` 3.0.0. `AbstractAsyncChannel` defines the generic transport interface parameterised over message type `M`; `AbstractChannel` is a synchronous specialisation that restricts all return types to concrete values. `RDACHannel` overrides `read()` to return a `Message` object rather than raw bytes. Green fill indicates classes introduced in this version; yellow fill indicates non-standard interface. 48
- 4.4 Protocol class hierarchy in `pyfgc` 3.0.0. `FFPProtocol` is the new addition that enables FGC4 communication; the other three protocols support existing device generations. Green fill indicates the class introduced in this version. 49

4.5	Class structure of the <code>pyfgc 3.0.0</code> connection layer. <code>AsyncConnection</code> and <code>AsyncConnectionFactory</code> form the async-first implementation; <code>SyncConnection</code> and <code>SyncConnectionFactory</code> are thin wrappers that delegate every operation to their async counterparts via <code>SyncWrapper</code> , providing a fully synchronous interface without duplicating any logic.	51
4.6	FFP subscription flow at the <code>AsyncConnection</code> layer. Registering a message callback starts a background task that dispatches every incoming message to the handler. After the <code>SUBEX</code> acknowledgement, <code>NOTIFICATION</code> messages pushed by FGC4 are received by the background task and delivered to the callback without polling.	52
4.7	FFP <code>GET</code> message flow at the <code>AsyncConnection</code> layer. The connection delegates serialization to <code>FFPProtocol</code> and transport to <code>AsyncWebSocketsChannel</code> . The two layers are fully independent and can be substituted without changing the connection code.	53
4.8	Class structure of the <code>pyfgc 3.0.0</code> <code>AutoConnection</code> layer. <code>AsyncAutoConnectionFactory</code> resolves the device name via <code>pyfgc_name</code> and selects the appropriate channel and protocol pair (WebSocket with FFP for FGC4, TCP/IP with NCRP for FGC3 and earlier). <code>SyncAutoConnection</code> owns an <code>AsyncAutoConnection</code> and delegates all operations via <code>SyncWrapper</code> ; <code>SyncAutoConnectionFactory</code> delegates all factory calls to <code>AsyncAutoConnectionFactory</code>	54
4.9	<code>AutoConnection</code> creation and use. The RBAC token must be obtained by the caller beforehand via <code>pyfgc_rbac</code> ; it is passed to the factory and stored on the connection for use in subsequent requests. The factory resolves the device via <code>pyfgc_name</code> , selects the appropriate channel and protocol based on the device class, and returns a ready-to-use <code>AsyncAutoConnection</code> . The caller interacts only with the high-level <code>get/set</code> API.	55
4.10	<code>AutoConnection</code> creation and use for an FGC3 device. The factory selects <code>AsyncTCPIPChannel</code> with <code>NCRPProtocol</code> based on the device class. Authentication differs from the FFP path: instead of an <code>INIT</code> handshake, the RBAC token is written to the <code>CLIENT.TOKEN</code> property on the gateway before the <code>GET</code> request is issued on the device.	55
4.11	Automatic RBAC token renewal in <code>AutoConnection</code> . When an expired token is detected before the FFP <code>INIT</code> handshake, the connection transparently renews it via the RBAC server and attaches the fresh credentials to the handshake. The caller receives the property value without handling token expiry.	56

- 4.12 **SyncWrapper** dispatch. Starting the wrapper once spawns a daemon thread running a dedicated **asyncio** event loop. Each **run_sync()** call submits a coroutine to the background loop via **run_coroutine_threadsafe**, blocks the calling thread on the future result, and returns the value once the coroutine completes. 58
- 4.13 Class structure of **pyfgc_name** 3.0.0. **AbstractEntry** defines the common serialization interface; **Entry** provides a partial implementation; **Device**, **SubDevice**, and **Group** are concrete entity classes. **KeepDevice** represents devices in the *keep* operational category. **Names** aggregates entity instances and exposes loading and query methods. 59
- 4.14 Class structure of **pyfgc_rbac** 3.0.0. **RBACToken** is an immutable value object wrapping a decoded RBAC token; all authentication methods on **RBACManager** return **RBACToken** instances. **from_base64** and **from_packed** are static factory methods. **get_kerberos()** is new in this version. 61
- 4.15 Class structure of **pyfgc_statussrv** 2.0.0. All methods on both classes are static. **StatusServerClient** handles network communication and delegates binary decoding to **StatusDecoder**. The dashed arrow indicates a dependency: **get_status()** and **get_status_from_channel()** call **StatusDecoder.decode_status()** internally. 62
- 4.16 Request handling in FGC API 2.0.0. Each incoming HTTP request creates a dedicated **AsyncAutoConnection** through **AsyncAutoConnectionFactory** (see fig. 4.9) and destroys it in a **finally** block after the operation completes. This per-request connection model replaces the queue-based **FgcPoolWrapper** of version 1.x, allowing concurrent in-flight requests to different devices without serialisation. 64

4.17	Server architecture of FGC API 2.0.0. The <code>FastAPI</code> application exposes HTTP endpoints through a router layer, which delegates device operations to a CRUD layer. A <code>DependenciesManager</code> singleton owns a <code>Names</code> instance (refreshed every 60 s by a background task reading the CERN name files every 30 s), an <code>RBACManager</code> , and an <code>FgcConnectionsManager</code> that creates <code>AsyncAutoConnection</code> instances on demand bounded by a per-gateway semaphore of 64. A separate <code>FGCStatus</code> singleton polls device status via <code>pyfgc_statussrv</code> every 0.5 s from both the TN and GPN status servers. The GPN instance exists because development FGC4 boards reside on the GPN; relocating them to the TN would require physical switch migration, increased security procedures, and IT oversight of the lab. Since FGC4 firmware introduces support for publishing device status to the status server, a dedicated <code>FGCD_STATUS</code> instance was deployed on the GPN so that development FGC4 boards can publish their status without requiring TN access.	65
4.18	LHC control system context for FGC4 and FGCDv2. FGC4 boards reside in the Technical Network (TN); each cluster consists of one gateway board running FGCDv2 (the FFP WebSocket server) and up to three slave boards. Clients in the TN communicate via FFP directly using <code>pyfgc</code> or the RDA/CMW stack. The FGC API runs on a machine with access to both the TN and the GPN, bridging browser-based tools such as FGC Commander to the device layer.	70
4.19	Deployment topology of FGCDv2 on the FGC4 gateway board. FGCDv2 runs as a Linux process on two of the four CPU cores and exposes an FFP WebSocket server (<code>FfpServer</code>). The <code>FfpClient</code> class (introduced in this work, section 4.12.1) provides a C++ client for this same interface. The command layer dispatches requests to the FPGA, which handles I/O and fast operations and communicates with up to three slave boards over the fieldbus.	71
4.20	Class structure of <code>FfpClient</code> . The class inherits from <code>Component</code> for FGCDv2 lifecycle management and owns a <code>WebsocketClient</code> instance (composition). Green fill indicates the class introduced in this work; grey fill indicates an abstract base class from the FGCDv2 framework. <code>WebsocketClient</code> is a pre-existing FGCDv2 class.	72

- 4.21 Message flow for a GET request through `FfpClient`. The caller registers a message handler, then calls `sendGetRequest()`. `FfpClient` constructs a `requests::Get` struct and forwards it to `WebsocketClient` via `sendMessage()` and `std::visit`. When the server's JSON response arrives, `WebsocketClient` calls `onMessage()`, which deserializes the payload into a `ResponseVariant` and dispatches it to the registered handler. The response path is asynchronous: `sendGetRequest()` returns as soon as the message is sent, and the handler is invoked later on the network thread. 74

List of Tables

2.1	Principal FGC firmware device classes.	12
2.2	Principal FGCD gateway classes and the protocol each uses.	13
2.3	Fields of an RBAC authentication token [12].	21
5.1	Typed properties exposed by TestProperties in the mock protocol servers, with their initial value and accepted range.	79
5.2	Unit test counts and branch coverage by package (sprint branch, Python 3.14).	80
5.3	Summary of validation activities and results.	88

Listings

- 3.1 Top-level `get` function in `pyfgc 1.x`. A full session is created and torn down for every property access (`source/old/pyfgc-1.8.6/pyfgc/api.py`). . . . 32
- 3.2 Module-level globals in `pyfgc_rbac 1.x` (`source/old/pyfgc_rbac-1.7.0/pyfgc_rbac/pyfgc_rbac.py`). 33
- 3.3 Module-level dictionaries in `pyfgc_name 2.x` (`source/old/pyfgc_name-2.1.1/pyfgc_name/pyfgc_name.py`). Five separate globals hold the entire device database. 33
- 3.4 Module-level channel pool in `pyfgc 1.x` (`source/old/pyfgc-1.8.6/pyfgc/channel_manager.py`). All callers in the process share the same `_channels` dictionary. 34
- 3.5 `pyfgc_rbac 1.x` returns tokens as opaque bytes (top); `pyfgc 1.x` must decode the payload itself to determine expiry (`source/old/pyfgc-1.8.6/pyfgc/fgc_session.py`) (bottom). 35
- 3.6 Duplicated `get` implementations in `SyncProtocol` and `AsyncProtocol` (`pyfgc 2.x`). The logic is identical; only the `await` keyword and the internal channel call differ. 37
- 3.7 String-based protocol dispatch in `pyfgc 2.x` (`source/old/pyfgc-2.0.3/pyfgc/fgc_session.py`). 38
- 3.8 Nested plain-dictionary construction in `pyfgc_name 2.x` (`source/old/pyfgc_name-2.1.1/pyfgc_name/pyfgc_name.py`). The shape of each entry is implicit in the parse logic; there is no named type. 39
- 3.9 A representative function from `pyfgc_rbac 1.x` with no type annotations and a bare `except` clause. 40
- 3.10 Connecting to an FGC3 device in `pyfgc 1.x` requires the caller to manually orchestrate name resolution, token acquisition, and protocol selection (top). `pyfgc 3.x` reduces this to a single call (bottom). 42
- 3.11 The sole CI-passing test in `pyfgc 1.x` and `2.x` (`source/old/pyfgc-1.8.6/pyfgc/tests/test_ci.py`). An identical file exists in `pyfgc 2.x`. . . 43
- 5.1 Structure of the concurrent-request stress test against CFE-866-RDEV3. . . 86

Ringraziamenti

È sicuramente difficile ringraziare tutte le persone che mi sono state accanto in questo lungo, se non infinito, percorso. Dopotutto, i viaggi brevi sono anche quelli più noiosi. Oggi, a una manciata di giorni prima della consegna della tesi, mi trovo a scrivere queste righe.

Mi sono sempre ripromesso di non cascare in banali *cliché*, nei quali, ovviamente, sto finendo e finirò. In fondo, se non fossero veri, non sarebbero cliché. Temo scriverò anche troppo, perché non ho avuto l'occasione di farlo durante la mia laurea triennale (grazie Politecnico, per avermi permesso di cavarmela senza tesi!). Dovrò compensare anche per quei primi (non brevi) anni di università, in cui non ho avuto l'opportunità di ringraziare chi mi ha supportato e aiutato.

Il primo pensiero è di ringraziamento per la mia famiglia, che mi ha sempre sostenuto e incoraggiato (nonché foraggiato e supportato, non dimentichiamolo) in questo percorso, negli infiniti momenti di sconforto e di difficoltà, senza mai abbandonarmi. Sicuramente, senza di loro, non sarei arrivato a questo punto.

Subito dopo voglio ringraziare Zahraa, la mia bellissima e fantastica ragazza, che ha creduto in me anche quando ero il primo a non farlo. Onestamente, senza di te, non sarei riuscito a fare neanche un decimo di quello che ho fatto fin'ora (e tu me lo ricordi sempre). Ti spetta di diritto un *"Te l'avevo detto!"*.

Grazie infinite anche alla mia Relatrice, che con pazienza e gentilezza mi ha supportato e aiutato in questa tesi (rispondendo alle mie sporadiche email impanicate).

Adesso viene la parte difficile: non lasciare nessuno fuori. La parte che mi fa più paura.

Sono passati... troppi anni dal mio primo giorno in università. Quasi me lo ricordo ancora (un po' con terrore, a essere onesto). Da quel giorno sono entrato in un turbine infinito di persone, esperienze, amicizie, conoscenze (e persone che non ho sopportato, ma non può essere sempre tutto perfetto, dai) che hanno lasciato, in modi diversi, un segno indelebile in me. Se oggi sono quello che sono, lo devo a tutti voi. Molti di voi non li vedo (e non li sento, magari) da anni, ma sappiate che vi ricorderò sempre (ah, un altro cliché, lo so, ma

è la verità). La colpa è molto probabilmente mia, non sono mai stato bravo a mantenere i contatti.

Non citerò nomi, ma preferirò luoghi e momenti, rigorosamente in ordine sparso, perché sarebbe impossibile trovare spazio per tutti in queste poche righe. Ho avuto la fortuna infinita di incontrare un sacco di persone che mi sono state accanto, e sarebbe ingiusto citarne solo alcune, rischiando di dimenticare qualcun altro (e non voglio farlo, davvero).

Quindi vorrei veniste con me, in questo flusso di coscienza, fatto di ricordi, di momenti, di luoghi, di cose dette e pensate:

- C'è posto? Mi tieni un posto? Non trovo i quaderni, ho occupato mezzo tavolo con la giacca (e ancora me ne dispiaccio, perché non è mai stata una cosa carina da fare, in fondo)
- Gli uffici di Prévessin, i tremendi ristoranti francesi (R3), stasera O'Brasseur? Giochiamo a ping pong, i caffè di sotto al bar, R1 venerdì, Saint-Genis è...
- Harambe, che in qualche modo ha cambiato tutte le nostre esistenze in un fatale giorno di fine maggio del 2016. Mi hai aiutato più di quanto tu possa abbia mai potuto immaginare
- I momenti in cui dicevamo *"Ma ci credo che un giorno questo posto ci mancherà?"*, mentre ancora non sono sicuro che questa cosa avverrà (ma c'è ancora tempo, no?)
- Gli spazi del L26, con i pessimi caffè del bar, gli studenti della statale che occupano i posti, le lezioni delle 8.30, le code infinite durante le pause e le caramelle (meglio non specificare altro)
- I piantini (a volte reali, a volte figurati) mentre scrivevo la tesi o studiavo per gli esami
- I pomeriggi, e le serate, a casa di qualcuno a studiare, chi più e chi meno (io probabilmente meno, temo)
- I rituali pre- e post- esami (ricordo con nitidezza qualcuno che per scaramanzia apriva il sito per controllare l'esito rigorosamente sempre nello stesso posto, anche a costo di scendere dalla metropolitana per riprenderla nella direzione opposta)
- I tavoli dell'agorà, tutti distrutti dai taglierini di qualche architetto maldestro, in cui noi stavamo dall'alba al tramonto, tra gente che urlava, allarmi antincendio (peraltro sempre ignorati), armadietti occupati, una certa tessera del caffè, le pause pranzo con le partite a burraco

- Il mio supervisor, al quale temo di aver fatto passare qualche brutta giornata, che è sempre stata la persona più disponibile e gentile che potessi incontrare (grazie per avermi dato fiducia, Przemek!)
- L'ansia per gli esami, le lezioni (spesso non frequentate, mamma e papà), le centinaia di pagine di appunti scritti, studiati e condivisi, l'aggiornare nervoso i servizi online per vedere spuntare il voto dell'esame, alcuni voti scandalosi di cui ancora non mi capacito (nel bene o nel male)
- La BCL (che quando ho iniziato, ancora si chiamava BCA), con la sua ansia incalzante, la corsa mattutina per i posti, il terrore di non ritrovare il posto dopo una pausa prolungata un po' troppo
- La email del CERN, per la quale ho sudato freddo, l'incredibile tirocinio che ne è nato, tutte le persone che mi hanno accolto, aiutato, e sopportato (anche loro, sì) in questo anno e poco più
- La nave, i tavoli del trifoglio, le aule di architettura, l'edificio otto, il 26 (sì, di nuovo) all'assalto di quella fantomatica aula libera, l'unica di tutta il campus (e che, puntualmente, non era libera)
- Shannon, Cray, Dijkstra, Black, Knuth, i libri di testo, i paper (irriproducibili) e tutte le slides lette e sbobinate
- Le saltuarie incursioni a Bovisa, perché c'era sempre posto e l'aria condizionata era generosa, mentre ci lamentavamo di quanto fosse invisibile il campus Leonardo (salvo poi preferirlo sempre, per questioni prettamente geografiche)
- Quelle poche (purtroppo) volte in cui sono andato al botteillon, ma le tante (purtroppo?) serate passate al Moscow, quelle altre, tante, volte in cui "Stasera aperitivo?", io al Matricola non ci torno più, il bomba fragola, gli zaini rubati (per fortuna non il mio) alla spritzeria, colazioni e happy hour al Bancale

Mentre scrivo queste righe, non esagero, devo sopprimere qualche lacrima. *Sì, lo so, è un altro cliché.*

Grazie di cuore, grazie a tutti, a coloro che si rivedono in queste righe, e a coloro che magari queste righe non le vedranno mai. Grazie per tutto quello che avete fatto per me.