



Electronic Communications of the EASST
Volume 86 (1) Year 2026

deRSE26 - Selected Contributions of the 6th Conference for Research Software Engineering in Germany (Part 1)

Edited by Bernd Flemisch, René Caspart, Bernadette Fritzsch, Florian Goth, Jean-Noël Grad, Florian Thiery

Dual backend design for sustainable and mockable data communication software

Thibaut Oprinsen, Vincent Pollet, Pierre Aubert, Sami Caroff

DOI: 10.14279/eceasst.v86.1.2779

License: This article is licensed under a CC-BY 4.0 License.

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Dual backend design for sustainable and mockable data communication software

Thibaut OPRINSEN, Vincent POLLET, Pierre AUBERT, Sami CAROFF

Laboratoire d'Annecy de physique des particules, Université Savoie Mont-Blanc, CNRS/IN2P3, 9 Chem. de Bellevue, 74940 Annecy

Abstract: Scientific software for data acquisition and processing is often developed over extended periods of time, during which hardware, communication protocols, and requirements evolve. Careful software design and abstractions that enable easy updates and component replacement are therefore essential, in particular for *big science* experiments whose software must be maintained for decades.

This paper advocates for a *single-interface, dual-backend* design: the same interface is used to interact with a real communication backend used in production and a mock backend. The interface abstraction encapsulates the real backend, enabling easy maintenance and replacement, while the mock backend provides consistent, lightweight substitutes that are indispensable for fast unit and integration testing and simulating complex exchanges without any external infrastructure. Using a real and a mock backend simultaneously further allows real communication to be recorded and replayed later to reproduce complex scenarios for debugging or testing.

This design has been implemented in a collection of libraries called `phoenix` [P. a], which abstracts and mocks a clock and a generic socket interface. A mock backend and a ZeroMQ backend are provided in C++, Rust, and Python, all relying on a compatible serialization library. This basis enables the implementation of data processing pipelines that are multi-language, consistent, mockable, and well encapsulated, therefore easily maintainable and extensible. The development of these tools is funded by the Horizon Europe project OSCARS [Osc], with the goal of supporting diverse scientific use cases.

Keywords: scientific software, software architecture, API abstraction, backend design, mock testing, data acquisition, distributed systems, ZeroMQ

1 Introduction

Computer science is a rapidly evolving field in which software systems must continuously adapt to new requirements, technologies, and constraints. This need for evolution is particularly acute in scientific contexts, where software is not only a tool but also an integral part of the research process itself. Unlike industrial software, which is often developed with well-defined and relatively stable specifications, scientific software emerges from exploratory workflows: hypotheses evolve, experimental setups change, and new requirements arise as understanding deepens; consequently, the software is frequently modified, extended, or refactored over long periods of time. This is especially critical for large-scale scientific infrastructures—often referred to as *big science* experiments, such as particle physics facilities, astronomical observatories, or large distributed sensor networks—which are expected to operate for decades while both hardware and software environments evolve significantly.

Data acquisition and data processing systems, which typically process large amounts of data on distributed computing infrastructures that involve specialized hardware and low-level communication routines, are particularly concerned. Hardware setups frequently evolve due to upgrades or replacements, network speed and topology change accordingly, and third-party libraries evolve on their own, not necessarily following hardware evolution. The tight coupling between acquisition or processing logic and communication backends can lead to brittle designs, where any change in the underlying infrastructure requires substantial modifications across the code base.

Moreover, testing these systems is inherently costly: unit tests or small-scale simulations can verify isolated software components but fail to capture the complex interactions of the full setup, while reproducing realistic conditions may require access to specialized hardware or even expensive access to the actual experimental infrastructure. Certain failure modes or rare events—network congestion, sensor dropouts, timing anomalies—are difficult to reproduce on demand, and even when access is available, the scale of the hardware setup and the volume of data generated can make tests slow and difficult to manage. This cost is compounded by the iterative nature of scientific software: features are frequently added, modified, or discarded as new requirements emerge, making frequent validation both necessary and, with real infrastructure alone, impractical and/or prohibitive.

To address these challenges, it is crucial to adopt software architectures that promote modularity, abstraction, and testability. By introducing a well-defined application programming interface (API) that encapsulates communication details, it is possible to isolate the core logic of the system from the specifics of the underlying transport layer, enabling the substitution of different communication backends without requiring changes to the application code. The testing strategy is also a central concern: the use of mock components—lightweight substitutes that emulate the behavior of real systems—offers a powerful alternative to direct hardware tests, allowing developers to simulate complex interactions in a controlled and reproducible manner. However, implementing mock systems that accurately reflect real-world behavior while remaining easy to maintain is a non-trivial task.

In this paper, we advocate for a software design pattern based on a single API used with a dual-backend architecture. In this approach, the same API is used to interact with both a real communication backend, which provides access to actual communication mechanisms such as

network sockets or message brokers, and a mock backend, which simulates these interactions without requiring any external infrastructure. Real and mock backends can be used separately for production and testing purposes, respectively, but are always instantiated together. The advantage of this approach is the ability to simultaneously use real and mock components in hybrid configurations: interactions with a real system can be recorded through the mock backend and later replayed to reproduce specific scenarios, which is particularly valuable for debugging and regression testing.

To demonstrate the practicality of this approach, we present `phoenix` [P. a], a collection of modular libraries designed to help develop and maintain data acquisition and processing systems. The `phoenix` networking libraries [AO26a, P. e] implement the proposed dual-backend architecture and provide both real communication backends, based on technologies such as ZeroMQ [Hin], and mock backends for simulation and testing. The libraries are available in multiple programming languages, including C++, Rust, and Python, and rely on compatible serialization mechanisms to enable seamless interoperability. This multi-language support facilitates the construction of heterogeneous data processing pipelines that remain consistent, testable, and maintainable across different components.

The contributions of this paper are threefold. Firstly, we analyze the challenges associated with the long-term development and maintenance of scientific software, with a particular focus on distributed data acquisition and processing systems. Secondly, we introduce a dual-backend architectural pattern that addresses these challenges using adequate abstraction, enabling modularity and easy testability. Thirdly, we present the design and implementation of the phoenix libraries as a concrete realization of this pattern, illustrating how it can be applied in practice to support robust and extensible scientific software systems.

2 Context and Relation to Existing Mocking Approaches

The `phoenix` libraries aim to facilitate the development and improve the quality of data acquisition and data processing software, with a specific focus on modularity and long-term maintenance to support long-running scientific experiments. They are actively used in the real-time analysis [CAG+23] of the Cherenkov Telescope Array Observatory (CTAO), a system that presents the typical challenges of data acquisition and processing research software: asynchronously processing in near real time several GB/s of data from dynamically allocated sensor arrays, while deployed on a distributed infrastructure shown in Figure 1. It is a good example of a situation where a sustainable testing strategy is required: individual programs perform high-frequency ($\sim$ 10kHz) exchanges to aggregate data from several sources; therefore, analysis results depend on the order and speed of exchanges — typically influenced by network and computing environment — which cannot be captured by unit testing of individual components; integration tests are too costly to capture many complex behaviors and to run on laptops or in continuous integration in a reasonable time; access to the actual hardware is limited as telescopes are under construction or used for science observations while the software is being developed; the software is expected to operate over a period of several decades, which makes it likely that data format or communication protocols will be updated or replaced.

To adequately test a program that asynchronously receives data from several sources before

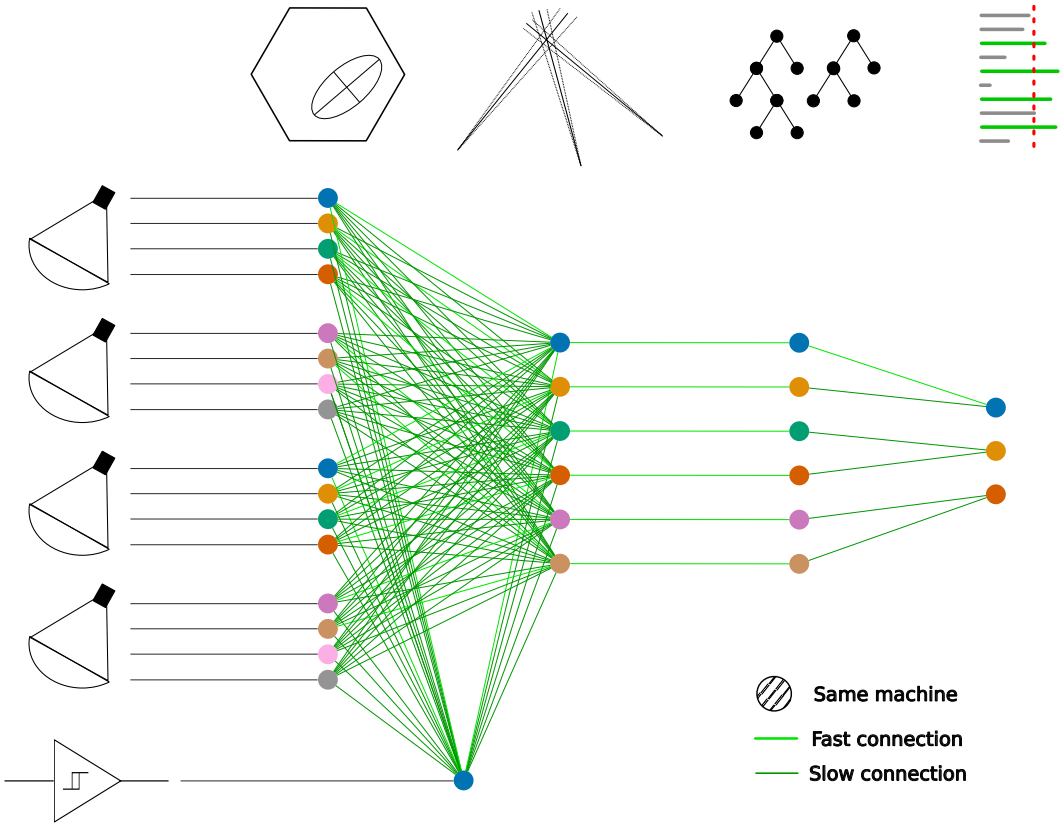


Figure 1: Diagram of the real-time analysis deployment of the Cherenkov Telescope Array Observatory, as a typical example of a distributed data processing system that requires an adequate test strategy. Colored dots represent processing programs, lines indicate data exchange between processes and columns indicate the analysis stage. From left to right: an array of sensors (telescopes and triggering system, 1st column) send data directly to programs (2nd column) performing a first processing step, after which data from several sources are gathered according to trigger information to perform stereo analysis (3rd column), followed by machine learning reconstruction models (4th column) and background rejection algorithms (5th column).

processing it, it is necessary to consider test scenarios for every possible interaction sequence of the program. The ”happy path” sequence (nominal operation sequence) of an example program with 3 interlocutors is shown in Figure 2. Testing even this happy path with real interlocutors may be unfeasible if those interlocutors involve actual hardware. Even considering only software interlocutors, precisely testing a single scenario with the real interlocutors is often very difficult: production programs are not designed to control when or in which order communications occur and can have random behavior. Tediously controlling start/stop times or introducing latencies for testing scenario purposes leads to slow and brittle tests, whose success depends on the speed of programs in a single specific deployment setup. Testing all different scenarios including error paths (missing communication, missing or unknown interlocutor, timeouts, etc.) is unsustainable

with only integration tests.

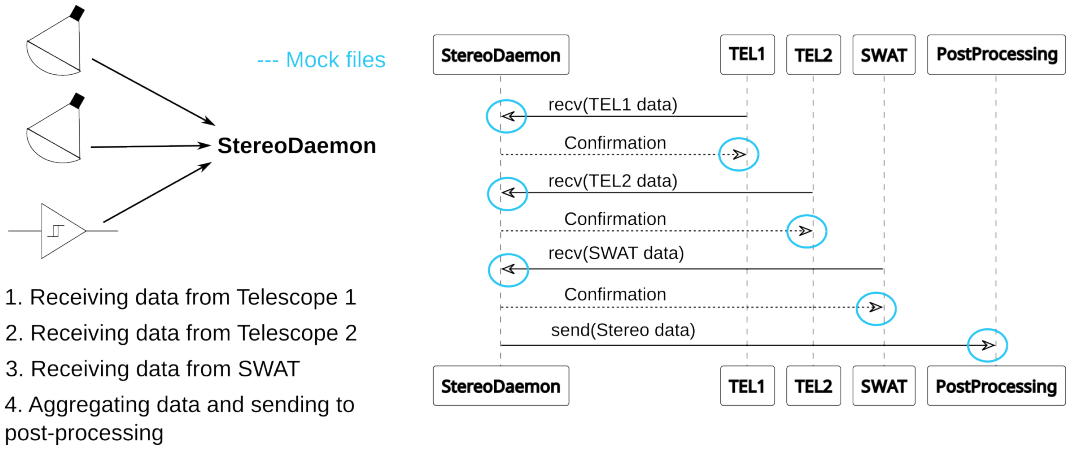


Figure 2: Unit testing for stereoscopic analysis with 2 telescopes and 1 trigger system.

The use of mock objects, test doubles, and record/replay testing is a long-established practice in software engineering [Fow07], and mature libraries already exist to support it. General-purpose mocking frameworks such as `GoogleMock` [Goo] (C++) or Python's `unittest.mock` [Pyt] let developers substitute arbitrary objects or functions with programmable stand-ins during tests. While this approach works well to test interfaces between objects within a program, one key limitation is that it is often difficult to write a *simple* mock object that mimics a complex external interlocutor. Implementing mocks for all nominal and error-path scenarios may require the mock object to be nearly as complex as the real interlocutors, or to implement a dedicated mock object for each scenario. In the context of distributed data acquisition or analysis software that undergoes frequent changes, the cost of implementing and maintaining the mock objects up to date with the real interlocutors makes this approach unsustainable.

Another strategy involves using mock objects at the level of the communication interface of the program to test: instead of mocking up an external interlocutor as in the previous approach, it is the communication interface inside the tested program which is mocked during tests. The goal is not to check the result of complete interaction with a simplified interlocutor, but rather to check that each usage of the interface with the interlocutor is as expected. With this strategy, testing a complex scenario amounts to checking that a sequence of interface usages is correct. This approach is much more sustainable: a single mock implementation of a communication interface can be used to test integration with all interlocutors using this interface, and to implement a test for a new scenario or a new interlocutor, it is only necessary to provide the expected sequence of events, which can be manually crafted or recorded from real interactions. This strategy is, for instance, applied to the HTTP requests interface by the `WireMock` library [Wir] (Java) or the VCR libraries [Mar10] in Ruby, which inspired many ports in other languages [Mc]. An additional advantage of this approach is that tests are faster and require fewer resources since there is no need to run interlocutors at all, even mocked ones: the integration is verified directly in the tested program.

The dual-backend design pattern presented in this paper explicitly applies the interface mock-

ing strategy: the interface API is a contract that must be fulfilled by both real and mock backends. The mock is a first-class implementation of the interface and not just a test-only product, and it can reliably be used not only to test but also to develop software using the API without ever using the real objects. This pattern can be applied to any interface, in particular to custom, low-level or multi-language interfaces that are common in scientific data acquisition and do not have dedicated and mature mocking libraries covering the use case. The systematic and consistent usage of this pattern in the `phoenix` libraries, demonstrated with the socket API, results in modular, interchangeable, and well-tested implementations, consistent across multiple languages, which is not covered by existing mocking libraries. In addition, this design forces the encapsulation of the communication technology and encourages the development of modular and reusable backends, both aspects that are desirable for long-term maintenance of research software.

3 Design Overview

The architecture proposed in this work is based on three abstractions: an *abstract* API, a *real backend*, and a *mock backend*. In the context of data transfer described in this work, the design pattern is applied to the Socket concept:

- **abstract API**: defines the API through which *any* socket will be used. This API standardizes the socket functionalities (e.g., `send`, `recv`) and parameters (e.g., `blocking`, `timeout`).
- **real backend**: a real backend must implement the abstract API, but is allowed additional, backend-specific parameters. For instance, Unix domain sockets [uni] and ZMQ [Hin] sockets offer the abstract socket API functionalities, but have additional concepts and parameters.
- **mock backend**: a mock backend must also implement the abstract API (`AbstractSocket`), but it must additionally implement the mock API (`AbstractSocketMock`) specific to mock-ups. In our case, mock sockets write and read their mock messages using a binary file; the mock API therefore defines the configuration for interacting with these files: which file to use (`set_prefix`) and whether to record or replay (`set_is_record`).

Built on top of this API, a Generic Socket Manager manages a collection of Generic Sockets, each instantiated according to the previously defined API. Software using sockets does not interface directly with the mock or the real backend; instead, it interfaces with a *generic* socket that only exposes the *abstract* API. The generic socket instantiates *both* a mock and a real backend, and dispatches the abstract API calls to the backends at run-time depending on the operating mode (see Section 3.5). The relationship is illustrated in Figure 3.

This design achieves *complete* encapsulation of the backend in a single object that can be easily replaced by any other implementation that fulfills the abstract API. The mock-up capabilities are encapsulated in separate backends; therefore, changing the real backend does not require the development of a new mock-up. Similarly, mock backends can be changed with no impact on the real backend.

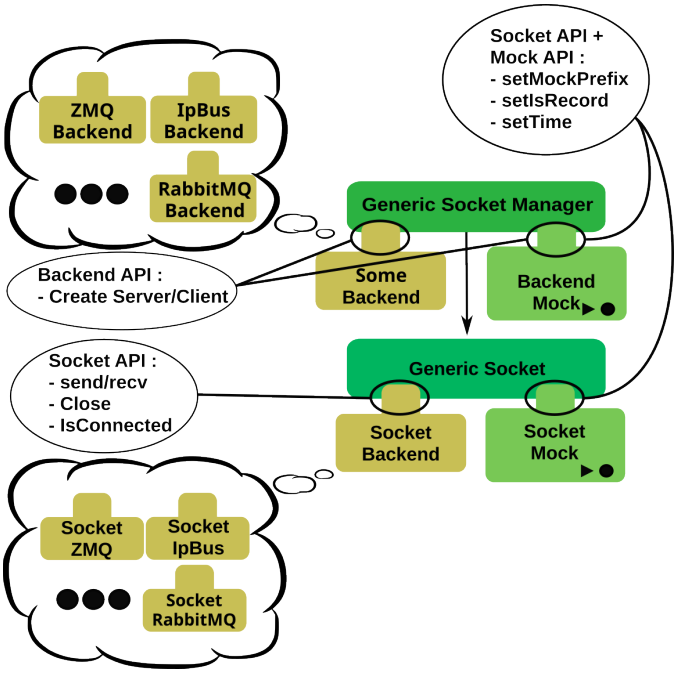


Figure 3: API overview: a socket manager instantiates a mock and a real backend and uses them to handle a collection of generic sockets, which are only interacted with through the abstract API. Generic sockets instantiate both a real and a mock backend and dispatch API calls depending on the operating mode.

The `phoenix` libraries [P. a] implement this design, providing a generic socket abstraction [AO26a, AO26b], a mock socket serializing/deserializing arbitrary messages to a binary file, and a ZMQ real backend. Consistent implementations in C++, Rust, and Python are provided.

3.1 API Abstraction Layer

The central element of the implementation is the class template `PGenericSocket<_TBackend, _TMockBackend>`, which provides the *single, standardized interface* for all socket operations regardless of the underlying transport technology. Application code interacts exclusively with this template through four primitives: `sendData` / `recvData`, which serializes / deserializes the input objects before transferring them with the socket, and `sendMsg` / `recvMsg`, which operate on already serialized data.

The template parameters `_TBackend` and `_TMockBackend` are specified at compile time; therefore, a single source file needs to be modified to switch from ZeroMQ backend to a Unix-socket backend, or any other custom backend. The application logic that uses the abstract API remains unmodified, and a mock backend can operate independently of the chosen real backend.

A `GenericSocket` object instantiates and owns a real and a mock Socket object, which are tied to its lifetime. The backend objects (mock and real), on the other hand, are owned by the higher-level caller and typically have a lifetime of the duration of a program.

When multiple sockets must be managed together, the class template `PGenericSocketManager<_TSocketKey, _TBackend, _TMockBackend>` provides a `std::map` registry that stores, creates, and routes messages to named `PGenericSocket` instances. Both the real backend instance and the mock backend instance are created and owned directly by the manager, ensuring a shared lifetime that is consistent with the lifetime of the managed sockets.

Generic (standardized) connection parameters are decoupled from the backend type through the plain structure `PSocketParam`, which gathers only the network-level information common to every transport: hostname, port, and optional send/receive timeouts. Backend-specific parameters (e.g. ZMQ socket type, or file path prefix for mocks) are encapsulated in a separate `Param` structure defined by each backend independently. This separation ensures that changing the transport layer has no ripple effect on the connection bookkeeping logic or the mock-up usage.

3.2 Backends API

Any concrete backend—whether real or mock—must satisfy the same structural contract, which enables the abstract layer to compose them interchangeably:

- A **socket class** that exposes multiple API functions, such as `sendData`, `recvData`, `createClientSocket`, `createServerSocket`, `isConnected`, and `close`. It must also declare two public type aliases: `Message` (the on-the-wire serialization type, typically a `DataStreamMsg` which is defined as a vector of `uint8`) and `Param` (the backend-specific creation parameters).
- A **generator class** that owns the backend context (e.g. the ZMQ context object) and delegates socket creation to the socket class. The generator also exposes a small set of static helpers (`msgResize`, `msgSize`, `msgData`, `msgToMock`, `mockToMsg`) that adapt the native wire message to and from the `DataStreamMsg` type used by the mock layer.

This two-class pattern (socket + generator) cleanly separates *per-socket state* from *backend-wide context*, a distinction that is particularly relevant for backends such as ZMQ where a single context must be shared across all sockets of the same process.

3.3 Real backend

The real backend implements genuine network communication and requires access to a live network or inter-process communication channel. In the `phoenix` library collection, the reference real backend is based on ZeroMQ [Hin], a high-performance asynchronous messaging library widely used in scientific computing. Because its abstract interface is identical to that of the mock backend, any real backend can be substituted for another without touching the application code, which directly addresses the long-term maintainability goal.

3.4 Mock backend

The mock backend provides a fully self-contained simulation of socket activity. Two implementations are provided in `phoenix`:

- **`PMockSocketGenerator`**: an in-memory mock that accumulates messages in a vector during the lifetime of the socket and flushes the data to disk upon destruction. This is the preferred choice for unit tests because it introduces no I/O latency during message passing.

- **`PMockSocketFileGenerator`**: a file-backed mock that persists each message to disk immediately on every send or receive operation. This variant is more I/O-intensive but guarantees that test data survives a process crash, making it suitable for debugging scenarios where developers must investigate the exact sequence of exchanged messages post-mortem. A dedicated command-line utility, `phoenix_mock_restore` from the *PhoenixMock* library, can repair a partially written mock file after an abnormal termination.

In both cases, the mock backend reads from and writes to mock files whose names are derived from a configurable prefix combined with the socket's hostname and port. This naming convention makes each mock file uniquely associated with a logical communication endpoint, so that a test suite can ship a set of pre-recorded mock files alongside the test code to reproduce complex interaction scenarios deterministically.

3.5 Operating Modes and the MOCK_RECORD Mechanism

`PGenericSocket` exposes a `setMode` method that switches the socket between three operating modes at any point during the lifetime of the object:

`NO_MOCK` All send and receive operations are handled solely by the real backend. The mock backend is instantiated but never invoked. This is the production mode.

`MOCK` All operations are handled solely by the mock backend. The real backend is instantiated but never invoked, so no network infrastructure is required. This mode enables fully offline unit and integration testing.

`MOCK_RECORD` Both backends are active simultaneously. On a send operation, the message is forwarded to the mock backend first; if the mock send succeeds (written to disk), the message is then forwarded to the real backend. On a receive operation, the real backend is read first and its return status propagated to the caller; the received data is additionally forwarded to the mock backend, which records it to disk. This mode allows developers to *record* real communication sessions transparently and reuse the records as deterministic test inputs in subsequent `MOCK` runs.

The three modes enable easy iteration between production and testing: real traffic recorded in `MOCK_RECORD` mode can be replayed offline in `MOCK` mode to reproduce and diagnose issues, without ever requiring access to the live system. This is particularly valuable in experimental scientific contexts where access to the real hardware may be scarce, shared among several teams, or simply unavailable between acquisition campaigns.

3.6 Summary

The application sees only the `PGenericSocketManager` and the `PGenericSocket` interface. Below that interface, the generic socket holds one instance of a real socket and one instance of a mock socket, both of which share the same `Message` and `Param` type vocabulary imposed by the backend APIs. The active mode determines at run-time whether the data should be processed by the real instance, the mock instance, or both. New real backends—for example, a future RDMA or shared-memory transport—can be plugged in by implementing the two-class (backend and socket) API contract, without modifying existing application code or mock functionality.

4 Use Cases and Evaluation

4.1 Single Process Unit Testing

One of the primary motivations for the dual-backend design is the ability to run unit tests without any network infrastructure, using a single process and without latency. A typical test suite for a component that produces and consumes messages involves two logical roles: a *sender* (e.g., named `Alice`) and a *receiver* (e.g., named `Bob`). In `MOCK` mode, both roles are served by the mock backend, which reads from and writes to files named after the socket endpoint (hostname and port). Because no real network is involved, the test executes entirely in a single process, completes in milliseconds, and produces no external side effects.

The following excerpt illustrates how the mock files can be easily created to be included in a test suite:

```
typedef PGenericSocketManager<std::string,
        PEmptySocketGenerator,
        PMockSocketGenerator> SocketManager;

// Sender side : record mode writes to mock files
SocketManager sender(PSocketMode::MOCK_RECORD);
sender.addServerSocket("Alice",
    PSocketParam{"localhost", 3390},
    Backend::server(), "", Mock::server());
sender.sendData("Alice", 42lu);

// Receiver side : mock mode reads from the same files
SocketManager receiver(PSocketMode::MOCK);
receiver.addClientSocket("Bob",
    PSocketParam{"localhost", 3390},
    Backend::client(), "", Mock::client());
size_t value = 0;
receiver.recvData("Bob", value); // value == 42
```

The communication occurring in the test is first recorded with the `MOCK_RECORD` mode of

the sender. This creates a file that can be included in the sender's test suite: running the same code in `MOCK` mode will assert that the sender's behavior is correct. In addition, the same file can be included in the receiver's test suite: running the receiver code in `MOCK` mode will read the file and assert that the receiver gets the correct information. Each test suite is self-contained: the mock files that serve as expected inputs are generated once (by running the sender in `MOCK_RECORD` mode against a live backend) and then committed alongside the test code of the sender and receiver, so that subsequent CI runs need only the `MOCK` mode and no network at all.

4.2 Simulation of Complex Multi-message Scenarios

The two-step test logic—first record, then replay—is used to create all test suites shipped with phoenix backend libraries: covering simple data types, raw messages, error cases, and file-backed mock variants. Since test scenarios are easy to create, high coverage of user code is easily achievable. The mock backends are, for instance, used to test the `PhoenixSocket` generic sockets: their introduction made it possible to raise the statement coverage from 65% to 91% without requiring a live real socket.

This is particularly useful for real-world application testing: verifying the processing of sequences of heterogeneous messages—control commands, configuration items, bulk data frames—or reproducing a representative scenario deterministically only requires writing the corresponding mock files. When the sequence can be performed using the application directly, running the application in `MOCK_RECORD` mode can be used to create the mock files. Otherwise, one can programmatically create the mock files by performing the sequence of calls to the socket interface directly. This approach is used to create the mock files of the real-time analysis of CTAO and is particularly useful to test the different scenarios of the data aggregation programs (as shown in Figure 2). In this case, this approach is the only way to implement such tests, as the observatory is still under construction and the hardware is not yet available for actual integration tests.

4.3 Reusable backend implementation

Adding a new backend to PhoenixSocket typically requires under 300 lines of code, split across a single header and a single source file that implement the two required classes—the socket class and the generator class—exposing the abstract Socket API described in Section 3.2. Backend-specific tests (e.g., exercising the ZMQ or Unix-socket transport itself) are kept in their own dedicated projects, so that PhoenixSocket's test suite only needs to validate the generic dispatch and business logic once, against the mock backend, independently of how many real backends exist.

4.4 Fast and reliable latency tests

In terms of raw execution time, simple tests running with the ZMQ backend and running with the mock socket backend are of comparable order (a few milliseconds). The mock backend's main benefit is removing the need for network infrastructure and the need for additional processes, rather than raw speed.

However, tests of behavior involving latency constraints (typically behavior in case of timeout) can be greatly sped up by applying the same dual-backend design to time management: `PhoenixSocket` can work concurrently with `PhoenixClock` [P. c], which provides a real and a mock clock backend, following the same design pattern. This makes it possible to test timeout-handling logic without waiting on wall-clock time: exercising a blocking send that never receives its confirmation would normally require waiting out the real configured timeout (e.g., 30 real seconds); with a mocked clock returning a scripted sequence of values when calling *now()* (e.g., 0, 1, 31), the same 30-second timeout is simulated and observed by the application logic in a few milliseconds, while still fully validating the timeout-handling logic. Such a test is illustrated in Figure 4. In the context of the CTAO real-time analysis presented in Section 2, the full test suite of the high-frequency, data-exchanging programs, including the timeout logic tests, runs in a few dozen milliseconds.

Mocking time also has the additional advantage of making the test of the logic independent of the execution time in the test setup: the test suite is robust across setups. This proves particularly useful in the context of applications like the real-time analysis of CTAO, which is deployed and tested in several environments.

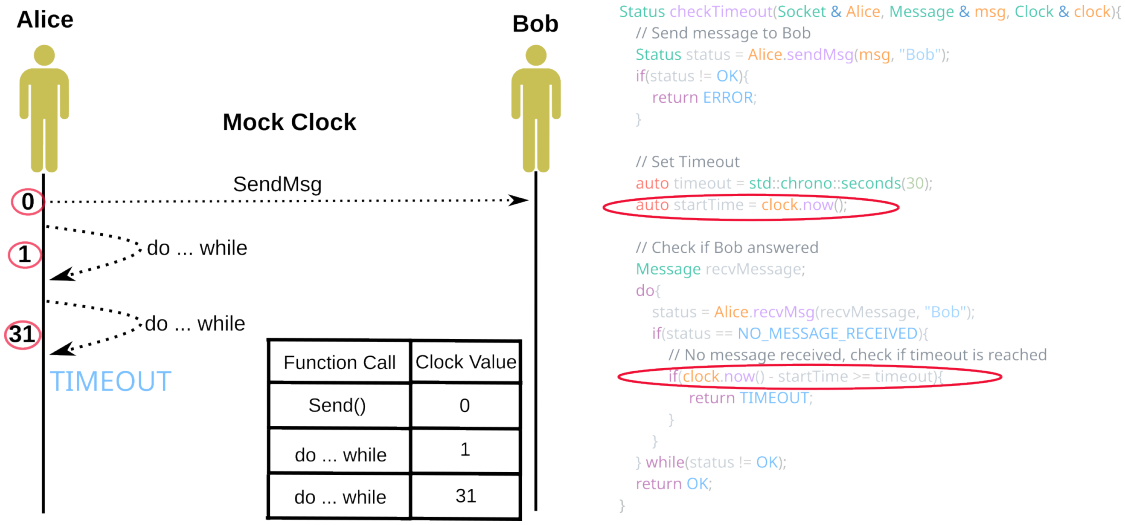


Figure 4: Mocking time in addition to communication: a way to test timeout logic without relying on actual execution latencies.

4.5 Record and Replay for Network Debugging

In a production scenario where a subtle communication bug manifests only under specific timing or data conditions, reproducing the failure in a controlled environment is often the hardest part of the debugging process. The `MOCK_RECORD` mode addresses this directly. By switching a deployed component to `MOCK_RECORD`—without any code change, only a configuration change—every message exchanged on the real backend is simultaneously recorded by the mock backend. After the failure has been observed and the mock files collected, the scenario can be

replayed offline in `MOCK` mode, where the developer can attach a debugger, insert additional logging, or modify the application logic and verify the fix, all without access to the live system.

This capability is particularly valuable in data acquisition (DAQ) contexts, where the real communication backend is typically connected to hardware that is only accessible during restricted operation periods. Being able to capture communication sessions during these periods and replay them offline at any time enables efficient and cost-effective iterative debugging, without requiring continuous access to the physical infrastructure. During the development of the real-time analysis of CTAO, this feature was often used to record data streams after updates to the sending sensors, for debugging and inclusion in future regression tests.

4.6 Multi-language and Multi-platform Pipelines

Because all phoenix libraries rely on the same `PhoenixDataStream` serialization format for the `DataStreamMsg` type, a C++ sender component and a Python or Rust receiver component can exchange messages transparently. The mock files thus generated by a C++ process can be replayed in a Python test, or vice versa, enabling cross-language integration tests that are as lightweight and reproducible as single-language tests.

5 Discussion

5.1 Advantages

Maintainability The strict separation between the API and the backend implementations means that replacing a communication technology—for example, upgrading from a version of ZeroMQ to a shared-memory transport—requires implementing only the two-class backend contract described in Section 3. Application code, test code, and mock infrastructure are entirely unaffected. The history of the PhoenixSocket library illustrates this: several breaking changes to the backend API (versions 4.0, 5.0, 6.0) were implemented without effort because the `PGenericSocket` abstraction isolated users from the backend details.

Extensibility The template-based design imposes no inheritance hierarchy and no virtual dispatch on the hot path in production. Adding a new backend is a purely additive operation: no existing files need to be modified. The `PEmptySocketGenerator` provided in the library serves as a minimal compilable skeleton that developers can copy and fill in to implement a new transport backend.

Testability The three operating modes (`NO_MOCK`, `MOCK`, `MOCK_RECORD`) enable a very fast cycle between testing and production. New features can be developed entirely in `MOCK` mode on a laptop, deployed in `MOCK_RECORD` mode for a first integration with real hardware, and finally run in full `NO_MOCK` mode in production—all with the same executable, differing only in the mode configuration. Mock files that capture edge cases or unexpected behavior can be recorded during `MOCK_RECORD` runs and naturally accumulate to form a growing regression test suite that captures real-world behavior.

Reproducibility and Open Science Because mock files encode communication scenarios in a binary format, they can be archived alongside publication data, enabling other researchers to reproduce the exact test conditions of a published experiment, even in different programming languages. This is directly aligned with the open science objectives of the OSCARS Horizon Europe project [Osc] that funds part of this work.

5.2 Limitations

Abstraction overhead Introducing an abstract socket layer adds a small layer of complexity between the application and the transport. In `NO_MOCK` mode, the mock socket is instantiated but never called, so the runtime overhead is limited to the construction and destruction of the mock object. However, in `MOCK_RECORD` mode every send and receive operation involves both backends sequentially, which introduces a possibly non-negligible overhead for the serialization and I/O operations. As a result of this overhead, a timing-dependent bug occurring in `MOCK` mode may not manifest in `MOCK_RECORD` mode, preventing the use of the mock files for debugging. Performance-critical programs should be profiled before enabling `MOCK_RECORD`.

Initial design complexity Adopting the dual-backend pattern requires some upfront investment: developers must define the two-class backend APIs, choose an appropriate mock variant (`PMockSocketGenerator` vs. `PMockSocketFileGenerator`), and design the mock file naming convention. For small, short-lived projects, this overhead may outweigh the benefits. The pattern is most valuable for systems expected to evolve over years and to be tested continuously in CI pipelines.

Mock fidelity The mock backend faithfully reproduces the *data content* of a communication session but does not reproduce *timing*, *latency*, or *concurrency* characteristics. Tests based solely on mock playback therefore cannot detect race conditions, buffer overflows caused by high-frequency bursts, or back-pressure behavior. Such properties must still be validated with real backends in integration or system-level tests, or a more realistic mock backend must be implemented.

6 Conclusion

Scientific software systems—and in particular data acquisition and processing pipelines for large-scale experiments—face a fundamental tension between their long development and operation timescales and the rapid evolution of the communication technologies and hardware they rely on. This tension is exacerbated by the cost and scarcity of access to actual experimental infrastructure, which makes exhaustive testing difficult.

This paper has presented a *single-API, dual-backend* design pattern addressing these challenges. By encapsulating communication operations behind a uniform generic socket interface parameterized by two backend types—a real backend for production and a mock backend for testing—the pattern achieves three goals simultaneously: it decouples application logic from

transport technology, enabling long-term maintainability and easy backend replacement; it enables fully offline unit and integration testing through lightweight mock files, reducing CI infrastructure requirements and shortening development cycles; and it supports record-and-replay workflows through the `MOCK_RECORD` mode, shortening the cycles between production and testing without requiring any code modifications.

The pattern has been implemented in `PhoenixSocket` [AO26a], a C++ library (also available in Rust and Python) that ships two mock backend implementations, a working empty backend skeleton, and a comprehensive test suite. The generic socket manager lets named sockets of different types coexist and be addressed uniformly, supporting realistic multi-channel pipeline topologies. The library is distributed as a conda package through the *prefix.dev* phoenix channel [pho].

7 Perspectives

Beyond its current implementation in `PhoenixSocket`, the dual-backend design pattern introduced in this work opens several promising directions for future developments, both in terms of application domains and software architecture.

Application to control frameworks The proposed software design is directly applicable to control system frameworks that rely on OPC UA [Fou], TANGO [Con], or EPICS [EPI], which are widely used in large-scale scientific infrastructures. In these systems, software components interact with distributed hardware through dedicated communication layers. Introducing a dual-backend design would enable offline simulation of control flows, facilitate testing without requiring access to the full control infrastructure, and improve long-term maintainability through standardized APIs.

Hardware communication protocols The pattern is also relevant for low-level hardware communication protocols such as IPBus. In such contexts, access to hardware is often limited or constrained by operational conditions. A mock backend capable of replaying recorded communication sessions would allow developers to test software behavior without requiring physical devices, significantly accelerating development and debugging cycles.

Robotics and embedded systems Robotics systems, which combine real-time constraints with heterogeneous hardware interfaces, could also benefit from this approach. Mockable communication layers would enable simulation of sensors and actuators, allowing developers to validate control logic and system behavior in a fully offline environment before deployment on physical platforms.

Device drivers and high-speed interfaces The dual-backend approach could be extended to the development of device drivers, including USB, optical fiber, or custom high-speed interfaces. By abstracting hardware communication behind a unified API, developers could validate driver logic using recorded or simulated data streams, reducing the dependency on specialized hardware setups during development.

REST APIs and distributed services Beyond socket-based communication, the same design principles can be applied to higher-level protocols such as REST APIs. Mockable HTTP clients and servers are already used in industry, but integrating them into a unified dual-backend architecture would provide stronger guarantees of consistency between production and test environments, especially in microservice-based systems, as well as cross-language mock compatibility.

Towards standardized and sustainable APIs More generally, this work highlights the importance of strict API abstraction in the design of long-lived scientific software. By decoupling application logic from communication backends, the dual-backend pattern promotes maintainability, extensibility, and reproducibility. Future work could explore the formalization of such patterns into reusable design guidelines applicable across scientific software domains.

8 Related Software

PhoenixSwarm The PhoenixSwarm project aims to support the easy implementation of long-running programs exchanging data via sockets. Programs own a single receiver socket, but can execute different processing functions based on the type of the received data. It relies on PhoenixSocket and PhoenixClock to handle communications in a maintainable and easily testable environment.

PhoenixDataStream The serialization layer used by `PhoenixSocket` for the `DataStreamMsg` type is provided by `PhoenixDataStream` [P. d], a standalone library for saving, loading, serializing, and deserializing arbitrary C++ data structures. This library has also been developed in Rust and Python. Future work will explore whether the `MOCK_RECORD` files produced by `PhoenixSocket` can be directly inspected and edited using the `PhoenixDataStream` toolchain, further easing post-mortem analysis.

PhoenixGenericMock The in-memory and file-backed mock handlers used internally by `PhoenixSocket`—`PGenericVecMock` and `PGenericFileMock`—are provided by `PhoenixGenericMock` [P. e], a library that generalizes the mock-file mechanism beyond sockets. Extending this library to support concurrent write access would allow several `MOCK_RECORD` sockets to share a single mock file, which would simplify the management of mock artifacts in large multi-socket pipelines.

PhoenixCheckStream `PhoenixCheckStream` [P. b] provides utilities for checking and validating data streams. Integrating stream-level assertions directly into the mock replay loop enables *in-line* verification of message invariants during test execution, further reducing the amount of test boilerplate developers need to write. It also provides human-readable feedback on test failure to indicate where in the mock message the error happens.

Acknowledgements: This work was conducted within the framework of the OSCARS [Osc] ESCAPE cluster, whose support and funding are gratefully acknowledged.

Data availability: All the Phoenix libraries [P. a] are available on Phoenix Community on Zenodo [P. a].

Declaration on the use of AI: Generative AI tools (large language model assistant) were used to help draft and revise portions of this manuscript's text, and to assist with parts of the implementation of the phoenix libraries (e.g., boilerplate code, refactoring suggestions). All technical content, design decisions, and results were reviewed and validated by the authors, who take full responsibility for the scientific and technical accuracy of this work.

Bibliography

[AO26a] P. Aubert, T. Oprinsen. PhoenixSocket. Mar. 2026.
doi:10.5281/zenodo.18839471
https://doi.org/10.5281/zenodo.18839471

[AO26b] P. Aubert, T. Oprinsen. RustyPhoenixSocket. Mar. 2026.
doi:10.5281/zenodo.19131742
https://doi.org/10.5281/zenodo.19131742

[CAG+23] S. Caroff, P. Aubert, E. Garcia, G. Maurin, V. Pollet, T. Vuillaume. The Real Time Analysis framework of the Cherenkov Telescope Array's Large-Sized Telescope. 2023.
https://arxiv.org/abs/2309.11679

[Con] T. Consortium. TAco Next Generation Objects.
https://gitlab.com/tango-controls/cppTango

[EPI] EPICS.
https://github.com/epics-base/epics-base

[Fou] O. Foundation. OPC-UA.
https://opcfoundation.org/about/opc-technologies/opc-ua/

[Fow07] M. Fowler. Mocks Aren't Stubs. 2007.
https://martinfowler.com/articles/mocksArentStubs.html

[Goo] Google. GoogleTest / GoogleMock C++ testing framework.
https://github.com/google/googletest

[Hin] P. Hintjens. ZeroMQ: High-performance asynchronous messaging library.
https://zeromq.org/

[Mar10] M. Marston. VCR: Record your test suite's HTTP interactions and replay them during future runs. 2010.
https://github.com/vcr/vcr

[Mc] K. McCarthy, contributors. VCR.py: Record and Replay HTTP Interactions.
https://github.com/kevin1024/vcrpy

[Osc] OSCARS Science Cluster.
https://oscars-project.eu/

[P. a] P. Aubert, T. Oprinsen, V. Pollet, S. Caroff. PHOENIX2 Library.
https://zenodo.org/communities/phoenix2/records?q=&l=list&p=1&s=10&sort=newest

[P. b] P. Aubert, T. Oprinsen, V. Pollet, S. Caroff. PhoenixCheckStream.
https://gitlab.in2p3.fr/CTA-LAPP/PHOENIX_LIBS2/serialize-io/PhoenixCheckStream

[P. c] P. Aubert, T. Oprinsen, V. Pollet, S. Caroff. PhoenixClock.
https://gitlab.in2p3.fr/CTA-LAPP/PHOENIX_LIBS2/clock/PhoenixClock

[P. d] P. Aubert, T. Oprinsen, V. Pollet, S. Caroff. PhoenixDataStream.
https://gitlab.in2p3.fr/CTA-LAPP/PHOENIX_LIBS2/serialize-io/PhoenixDataStream

[P. e] P. Aubert, T. Oprinsen, V. Pollet, S. Caroff. PhoenixGenericMock.
https://gitlab.in2p3.fr/CTA-LAPP/PHOENIX_LIBS2/serialize-io/PhoenixGenericMock

[pho] Phoenix Channel.
https://prefix.dev/channels/phoenix

[Pyt] Python Software Foundation. unittest.mock — mock object library.
https://docs.python.org/3/library/unittest.mock.html

[uni] Unix Sockets.
https://pubs.opengroup.org/onlinepubs/9799919799/functions/socket.html

[Wir] WireMock. WireMock: HTTP mocking and API simulation.
https://wiremock.org/