Concrete Architecture of GNUstep: A Detailed Analysis of libs-base

Alex Liang, Chaolin Zhao, Lucas Chu, Simone Jiang, Yueyi Huang, Zhongren Zhao
Queen's University
CISC 322/326 Software Architecture
Instructor*:* Bram Adams
TA: Mohammed Mehedi Hasan
Date*: 14/03/2025*

1. Abstract

This report presents a comprehensive analysis of the concrete architecture of GNUstep's CoreBase Library, building on our initial conceptual framework to examine subsystem interactions and code-level dependencies. By employing the Understand tool alongside extensive manual verification, we mapped the relationships among key components—including apps-gorm, libs-gui, libs-base, libs-back, tools-make, and libobjc2—to identify both expected and unexpected dependencies. Our investigation reveals that GNUstep's architecture embodies a hybrid model, blending traditional layered design with object-oriented principles. Notably, the study uncovers discrepancies such as absent direct links where anticipated and unforeseen couplings between modules, highlighting the dynamic evolution of the system over time. Through detailed reflexion analysis, we pinpoint how historical design choices and iterative development processes have led to divergences between the conceptual architecture and the actual implementation. Additionally, sequence diagrams and use case scenarios are used to illustrate practical workflows, providing further insight into the

operational intricacies of the system. Overall, our findings not only deepen the understanding of GNUstep's internal structure but also offer valuable lessons for managing and evolving complex, legacy codebases.

2. Introduction

2.1. Concrete Architecture of GNUstep

Large pieces of software systems like GNUstep often present significant challenges for developers due to their inherent complexity and intricate dependencies, especially those new to the project. In class, we created a conceptual architecture based on documentation and logical assumptions. However, developers should frequently verify and adjust their conceptual understanding to correct any inaccuracies throughout development.

This report provides a concrete analysis of GNUstep's architecture, examining specific subsystem interactions and code-level dependencies. Utilizing the Understand tool, we closely inspected GNUstep's codebase, particularly source files and headers, to map dependencies between subsystems.

2.2. Updated conceptual architecture in A1

"GNUstep is a cross-platform, object-oriented set of frameworks for desktop application development." (GNUstep Project, n.d.). In our prior analysis (A1), we concluded that GNUstep's conceptual architecture is organized into four cohesive layers to achieve cross-platform efficiency. While this layered approach effectively highlights modular separation of concerns, it does not fully capture GNUstep's object-oriented (OO) architectural foundations, which are deeply embedded in its design philosophy and implementation. The layered model emphasizes vertical separation (logic, UI, rendering), while the OO architecture focuses on horizontal cohesion through class hierarchies and design patterns. Together, they reflect GNUstep's hybrid approach: layers ensure cross-platform portability and modularity, while OO principles support maintainable, scalable codebases. Our updated analysis underscores that GNUstep's architecture is not merely layered but intrinsically object-oriented, with layered components implemented through OO abstractions. This dual perspective—modular separation combined with inheritance and polymorphism—offers a more precise view of how GNUstep balances portability, efficiency, and developer flexibility, guided by Objective-C's design paradigms.

2.3. Subsystem Analysis

Utilizing the Understand tool, we closely inspected GNUstep's codebase, particularly source files and headers, to map dependencies between subsystems. The key components include:

apps-gorm: A graphical interface builder primarily dependent on GUI components.
libs-gui: Manages interface elements, user events, and interactions.

libs-base: Supplies core utilities like memory management, file operations, and networking.
libs-back: Handles rendering graphical interfaces across different platforms.
tools-make: Oversees build and compilation processes, intended to remain isolated from runtime components.
libobjc2: Offers Objective-C runtime support.

While some of our initial assumptions held true, we also found some dependencies. For example, we found libs-base to be more fundamental than we thought, as it plays a critical role in supporting libs-gui and libs-back. In addition, we found that the link between libs-gui and libs-back is stronger than expected, since the GUI layer interacts directly with the rendering backend. Moreover, there is a running time dependency between tools-make and libobjc2 , which contradicts our assumption that tools-make runs independently. These findings emphasize the importance of validating conceptual models through concrete code analysis. This is because real software architectures often reveal hidden complexities that cannot be captured by documentation alone.

3. Explanation of architecture Derivation process

3.1. Virtualize demonstration of derivation process

The transition from Figure 1 to Figure 2 represents the derivation process in the architecture of the GNUstep project. This transformation is characterized by structural modifications in dependencies, optimizations, and updates in library interactions.

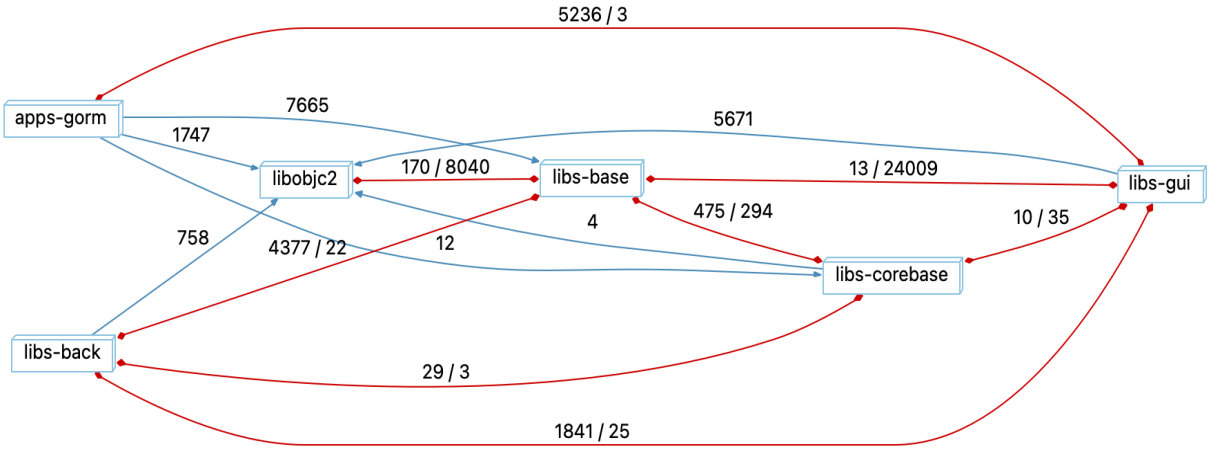


Figure 1: Dependencies between different subsystems based on directory structure

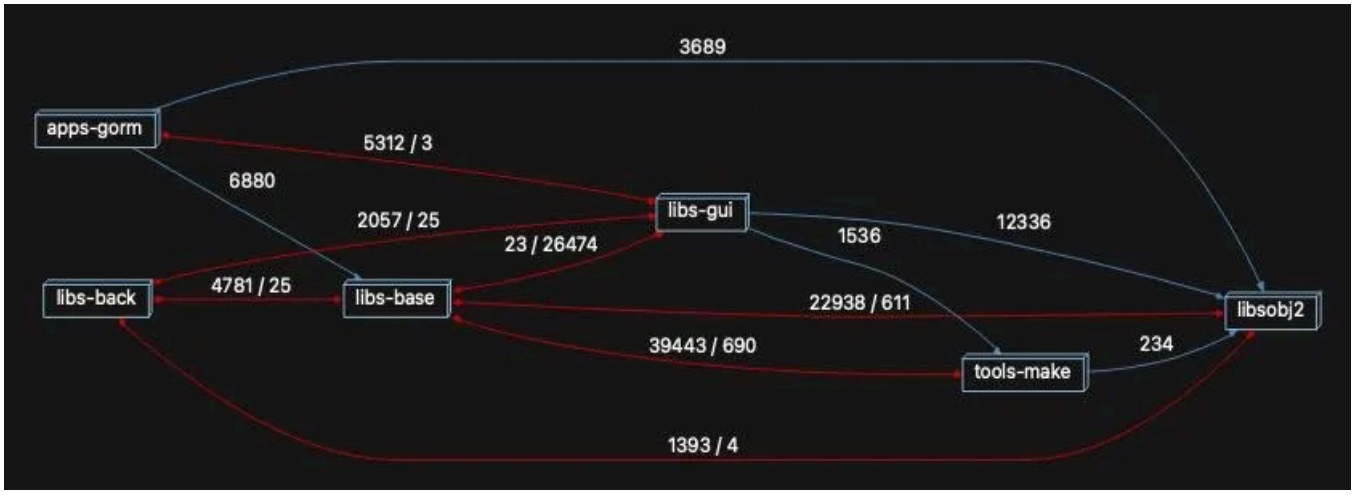


Figure 2: Dependencies between different subsystems based on concrete architecture

3.2. Methodology

The initial step in the process involved obtaining the source code for the GNUstep project which is provided in OnQ. Then download the required ***understand*** tool shown in the tutorial.

To help visualize module relationships and provide a foundation for verifying structural changes, we generate graphs in the *Understand* tool, using the given project file from onQ. Once the project is loaded, allow the tool to perform initial analysis to parse dependencies. Navigate to **Graphs** in the top menu and select **Dependency Graph - By Directory Structure**. You can refine the graph by selecting specific components, such as libraries or modules. Adjust settings to display call dependencies, file relationships, or function-level interactions.

3.3. Verification and Reflection

After completing the dependency verification, a reflection analysis was conducted. This involved comparing the anticipated architecture with the concrete system structure, highlighting key divergences. Several findings appeared, including the fact that libs-corebase was essentially replaced by pre-existing libraries, which resulted in its removal. Additionally, the weight of dependencies between certain modules had shifted significantly, indicating a reallocation of function call frequencies and module responsibilities.

Although Understand offered insightful information about dependence relations, accuracy was ensured through manual analysis in the final verification. In several cases, discrepancies between the manually mapped relationships and the automated outputs were discovered, highlighting the need of a thorough validation process.

4. Subsystems and its interactions

4.1. Reflexion analysis on libs-base

We compare the conceptual architecture defined in A1 with the actual architecture observed through the dependency graph generated by Understand. This analysis identifies consistent dependencies (those matching our initial design), divergent dependencies (those appearing unexpectedly in the code), and absent dependencies (those missing in the code despite being anticipated).

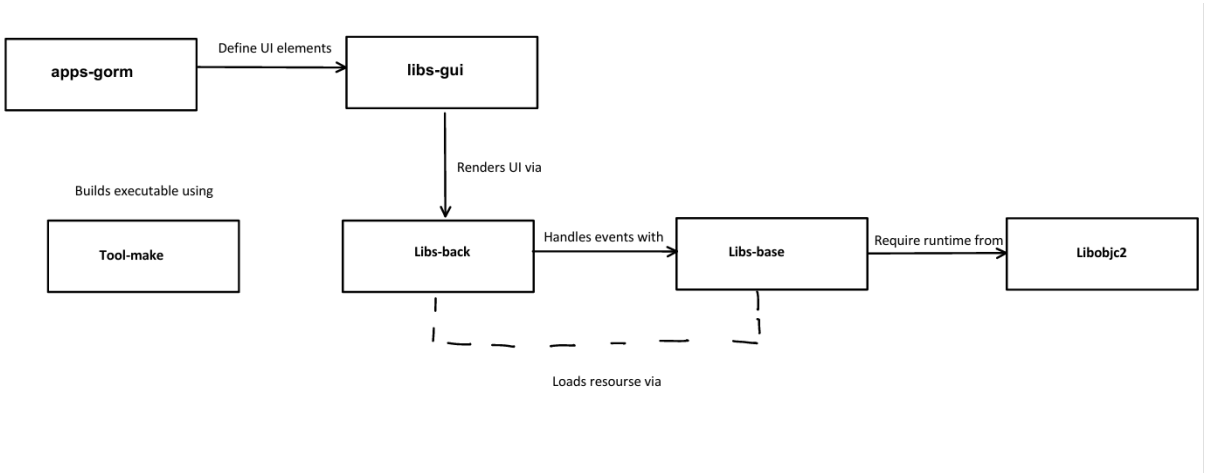
4.1.1. Recap of the Conceptual Architecture

In A1, we envisioned GNUstep's architecture as composed of the following key components:

libobjc2 (Objective-C Runtime): Underpins the entire GNUstep environment, providing runtime support for Objective-C features such as dynamic typing, message sending, and object allocation.
libs-base: Offers foundational functionality (memory management, file I/O, networking, collections, etc.). Acts as a platform-agnostic layer that higher-level libraries (GUI, back-end rendering) depend upon.
libs-gui: Manages user interface elements like windows, buttons, menus, and event-handling loops. Relies on libs-base for core services (strings, collections, memory handling).
libs-back: Handles low-level rendering and communicates directly with platform-specific APIs (e.g., X11, Windows GDI, Cairo). In the conceptual model, it should remain primarily dependent on libs-base, with minimal direct coupling to libs-gui.
apps-gorm: A visual interface design tool for GNUstep applications. Conceptually depends on libs-gui (to build UI components) and libs-base (for fundamental operations). Typically should not require direct interaction with libs-back unless previewing or rendering UI elements at a low level.
tools-make: A build system (GNUstep Make) and associated scripts, mainly responsible for compiling and linking.
Ideally, it should not form strong runtime dependencies with the libraries used in actual application execution.

From this perspective, **libs-base** should be the common foundation, **libs-gui** and **libs-back** should handle front-end logic and rendering respectively, and **Tools-Make** should only be involved in the build stage without strong runtime dependencies.

4.1.2 Box Arrow diagram



4.1.3 Reflexion analysis

Below is the resulting dependency graph (Figure 3.2) after manually labeling and grouping files in the GNUstep repository:

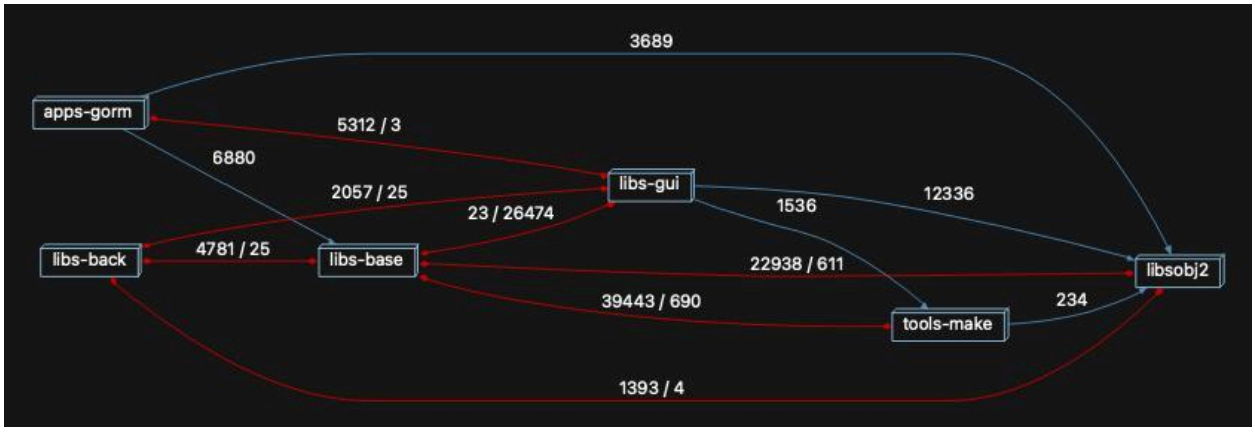


Figure 3.2 The dependency graph of the concrete architecture components based on the source code of the GNUstep project, generated from Understand.

From this graph, we did not observe any truly “unexpected” dependencies. Instead, the most prominent findings are absences, where certain links anticipated in our conceptual model are missing in the real codebase.

Absences

1. Lack of Direct Dependencies Where Expected

In our conceptual architecture, we anticipated that apps-gorm would communicate directly with libs-back for low-level rendering tasks—such as previewing interface layouts or handling advanced drawing. However, the Understand graph shows

no direct link from apps-gorm to libs-back, suggesting that any rendering operations are funneled exclusively through libs-gui. This implies a more streamlined pipeline, where Gorm depends on higher-level GUI abstractions rather than directly invoking the back-end layer.

2. No Apparent Build-Time Only Boundaries

We originally expected tools-make (or similar build scripts) to remain isolated from runtime libraries like libs-base or libs-gui, enforcing a clear divide between build processes and the application's execution environment. Yet the actual architecture does not strictly separate them: the dependency graph shows tools-make linking to libs-base. While this is not necessarily incorrect, the absence of a strict "build-only" boundary may complicate future refactoring efforts and cross-platform builds, since any changes in libs-base might inadvertently affect or break the build process.

Divergences

libs-base ↔ libs-back

libs-base unexpectedly references both libs-back and libs-gui. This is contrary to the conceptual model, where libs-corebase should remain independent of UI-specific code. Such dependencies may be due to code reuse (for logging, debugging, or error handling) or legacy refactoring artifacts.

plugins-themes-WinUXTheme → libs-base

plugins-themes-WinUXTheme shows dependencies on not only libs-back (as expected) but also on libs-base and libs-corebase. This indicates that theming logic is more deeply integrated into the core layers than originally designed.

Blurring of Build-Time and Runtime Boundaries

In our initial architecture, tools-make was expected to operate strictly as a build system, isolated from the runtime libraries like libs-base, libs-gui, and libs-back. Contrary to this expectation, the dependency graph indicates that tools-make is directly linked to libs-base. This divergence implies that certain build scripts or make utilities are invoking runtime functions—for instance, for file handling or logging purposes—thus blurring the intended separation between compile-time and execution-time environments. Such intermingling can complicate future refactoring efforts, as the coupling between build processes and runtime components increases the risk of unintended side effects when changes are made, potentially affecting cross-platform compatibility.

Direct GUI Dependency on the Objective-C Runtime

While our conceptual design posited that libs-gui would primarily rely on libs-base to access core system functionalities, the dependency graph shows that libs-gui also maintains a direct dependency on libobjc2. This was not explicitly foreseen, as we

expected the GUI layer to abstract away the intricacies of the runtime environment. The presence of this direct link suggests that parts of the GUI code are accessing low-level Objective-C runtime features—such as dynamic method resolution or class loading—without the intermediary of libs-base. This unexpected dependency increases the coupling between the GUI and the runtime, potentially complicating efforts to update or modify the underlying runtime environment without affecting the user interface layer.

Cross-Layer Integration of Theming Modules

The theming module, plugins-themes-WinUXTheme, was originally intended to interface solely with the rendering backend (libs-back), providing a means to adjust visual aspects without affecting core operations. However, the dependency graph reveals that this module also shows connections to libs-base and even libs-corebase. Such cross-layer integration indicates that theming functionality is more deeply embedded into the core infrastructure than anticipated. This can lead to a scenario where modifications in the core or base libraries might inadvertently alter the theming behavior, reducing the system's modularity and flexibility. This integration could be the result of shared utility code or legacy design decisions, and it highlights the need for a clearer delineation between visual presentation and core system logic.

Analysis of Causes and Impact

These divergences may result from historical evolution, where code reuse and gradual refactoring have led to blurred module boundaries. In some cases, performance optimizations or convenience in development may have introduced direct dependencies that contradict the original modular design. The impact of these issues includes increased coupling, reduced modularity, and potential maintenance challenges, as changes in one module may propagate unintended effects across the system.

4.2. Reflexion analysis of Networking 2nd level subsystem

4.2.1 Conceptual & Concrete Architecture

In this section, we examine the **Networking** subsystem in detail. Conceptually, Networking is responsible for managing connections, sending and receiving data, handling various protocols, and ensuring reliable communication within the system. It also coordinates with lower-level modules to handle concurrency (for asynchronous operations) and platform-specific APIs (for socket or network interface calls).

From a concrete standpoint (as seen in the dependency graph), Networking interacts with multiple modules:

core: Provides foundational services, such as object lifecycle and memory utilities.

platform: Supplies OS-level networking calls, which Networking may invoke directly (e.g., socket creation, event loops).

Concurrency & Threading: Offers thread pools or concurrency primitives for asynchronous I/O.

Misc and **Data & Time**: Potentially used for logging, timestamping requests, or error reporting.

This broader-than-expected interaction indicates that Networking is not just a simple “middle layer” but rather a subsystem that spans across concurrency, platform specifics, and core functionalities.

4.2.2 Reflexion Analysis

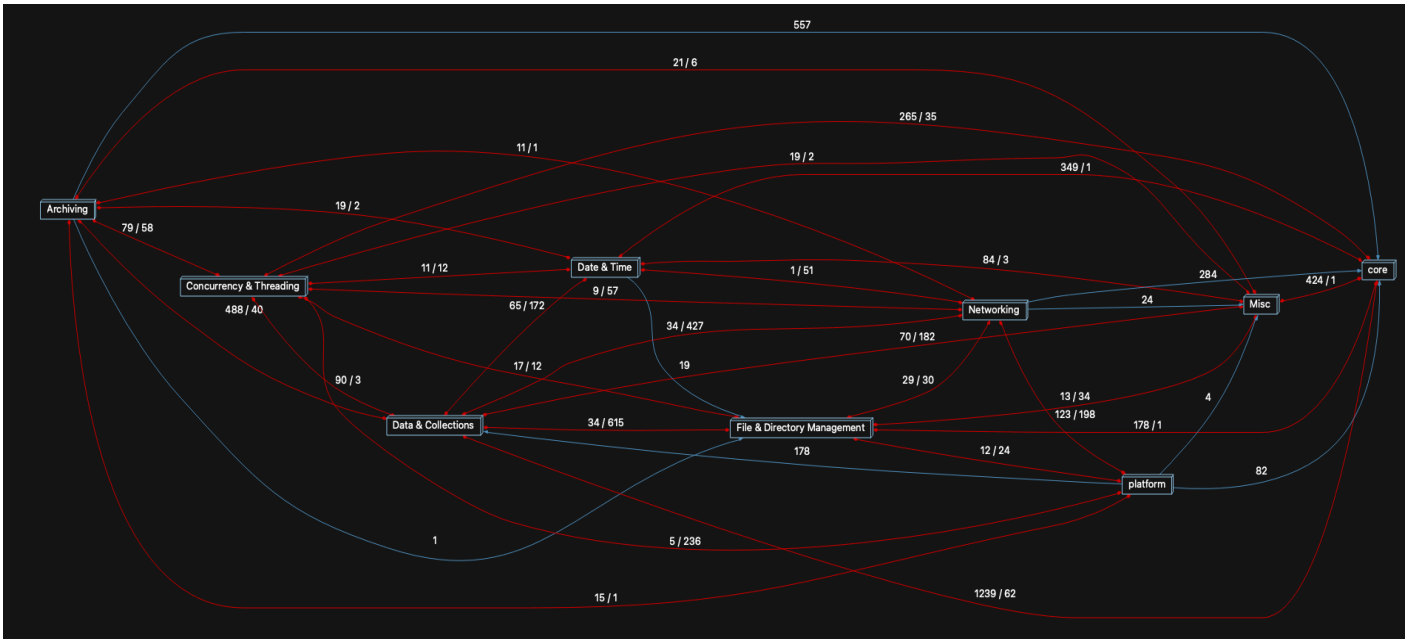


Figure 3.3 The dependency graph of the 2nd level concrete architecture components

Through closer inspection of the Networking subsystem, several noteworthy points emerge:

Absences:

Missing Interface with a Dedicated Storage/Caching Module:

The conceptual design anticipated a clear dependency between Networking and a separate storage or caching subsystem for tasks like cookie management. However, the dependency graph shows no such link, suggesting these functions have been absorbed elsewhere (likely within the Browser).

Lack of a Separate Security Module:

Instead of relying on an independent security component for SSL/TLS and certificate verification, Networking handles these aspects internally. This absence of a distinct security layer points to a consolidation of responsibilities that may undermine modularity.

Overall, the analysis shows that while the core functions of Networking are present, unexpected dependencies and missing interfaces result in a more intertwined architecture than originally designed. Addressing these issues through refactoring would help restore a cleaner separation of concerns and improve maintainability and scalability.

Divergences:

Excessive Coupling with Platform and Concurrency Layers:

Although the conceptual model expected Networking to use abstract interfaces for OS-level operations and asynchronous handling, the actual implementation directly calls platform-specific APIs and manages its own threading. This tight coupling complicates maintenance and portability.

Unexpected Cross-Subsystem Integration:

Networking unexpectedly interacts directly with modules such as Browser and Renderer—for error logging, network printing, and certificate handling—indicating that shared utility functions or debugging mechanisms have blurred the originally defined boundaries.

Direct UI Dependencies:

Parts of the UI directly invoke networking functions (e.g., for MIME type resolution), contrary to the envisioned layered approach where the UI should only use Networking through well-defined interfaces.

5. Use Cases

5.1. Sequence Diagram: client-server remote messaging

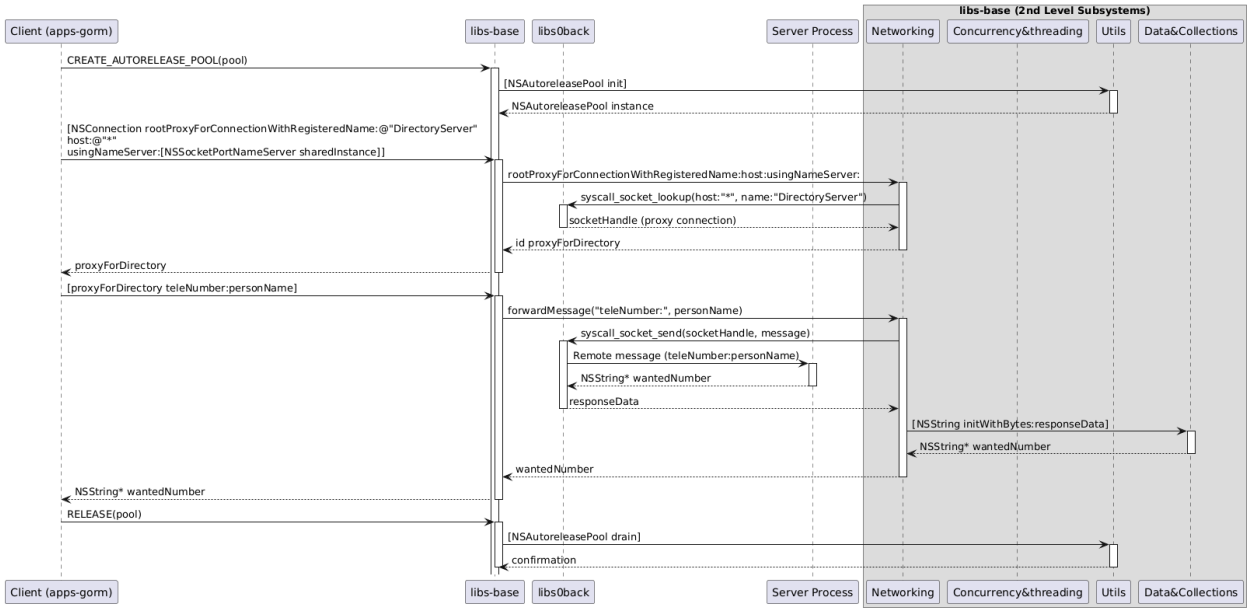


Figure 5.1: Sequence diagram of a telephone querying app on client side

Here we have an example from GNUstep Base Library Programming Manual(*Botto et al., 2004, p. 74*) that talks about client-server remote messaging using GNUstep. When the user begins the operation to retrieve a telephone number, the Client application initiates the process by forwarding the user's input to the libs-base subsystem. This subsystem oversees remote messaging and system coordination, ensuring a seamless workflow.

First, Proxy Acquisition, the Client initializes an NSAutoreleasePool through the Utils subsystem to manage memory efficiently and prevent leaks. The Client then requests a proxy for the remote server using the rootProxyForConnectionWithRegisteredName:host:usingNameServer: method, facilitated by the Networking subsystem. Networking delegates the socket lookup to the system-level libs0back component, which interfaces with the network stack to locate the server registered as "DirectoryServer" across all hosts.

Once the proxy is acquired, the Client sends the message [proxyForDirectory teleNumber:personName] via Networking, which forwards the request to libs0back. This component encodes the message and transmits it over the network to the Server Process. Upon receiving the request, the Server Process processes it and sends the result back. libs0back receives the response and relays it to Networking, which collaborates with the Data&Collections subsystem to decode and convert the raw bytes into an NSString instance.

In the end, the Client releases the NSAutoreleasePool via Utils, ensuring all temporary objects are deallocated, maintaining efficient memory management. The

retrieved telephone number is then returned to the Client, completing the operation successfully.

This workflow demonstrates how Networking for communication, Utils for memory management, and Data&Collections for data conversion works together to achieve remote, over devices communication on GNUstep applications.

5.2. Sequence Diagram: Storing a User-Input String

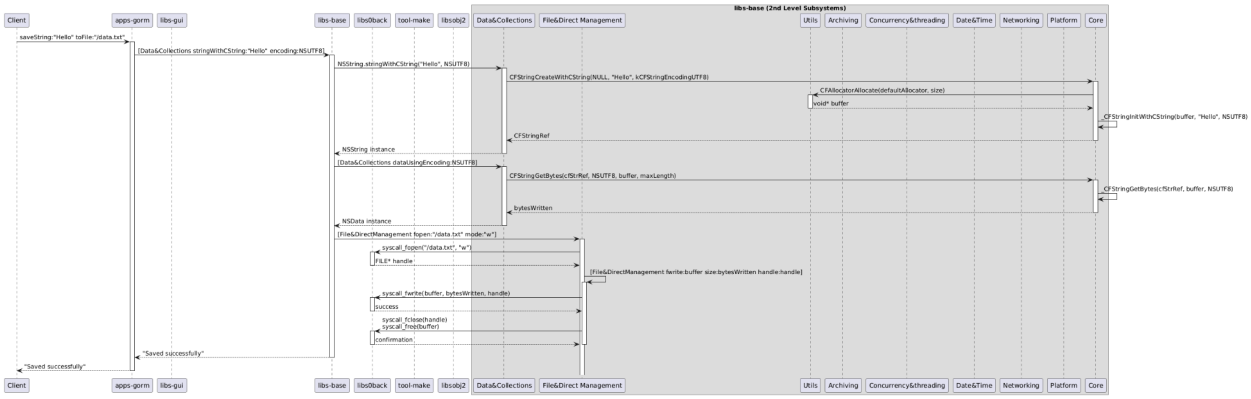


Figure 5.2: Sequence diagram of storing an user input process

The user begins by interacting with the Client application to save a string to a file. This request follows a structured workflow. When the user initiates the save operation, the Client forwards the input to the apps-gorm subsystem, which coordinates high-level tasks. apps-gorm delegates the string and file handling to the libs-base subsystem, a core library responsible for foundational operations.

Within libs-base, the request is routed to the Data&Collections subsystem, designed to manage strings and data conversions. Data&Collections processes the string creation by invoking the Core subsystem, which handles low-level encoding and memory allocation. The Core subsystem interacts with Utils to allocate a buffer for the string data, ensuring efficient memory usage. Once initialized, the string is wrapped into an NSString instance and returned to Data&Collections.

For file storage, libs-base delegates the task to the File&Direct Management subsystem, responsible for I/O operations. File&Direct Management coordinates with the libs0back subsystem, a low-level library that interfaces directly with system calls. libs0back opens the target file, writes the buffer to disk, and releases resources, ensuring compatibility with the operating system’s file management protocols.

Finally, a confirmation message is propagated back through the subsystems—libs0back to File&Direct Management, File&Direct Management to libs-base, and libs-base to apps-gorm—before the Client receives a "Saved successfully" notification. This workflow highlights the collaboration between libs-base’s specialized subsystems (Data&Collections, Core, Utils, File&Direct Management) and the system-level libs0back, ensuring reliable data handling and storage.

6. Lessons learned

Throughout our in-depth analysis of GNUstep's CoreBase Library, we gained several valuable insights that will inform future architectural evaluations. The key lessons learned are as follows:

Complementarity of Automated Tools and Manual Analysis

While the Understand tool provided useful dependency graphs and an overall architectural overview, its results served primarily as a guide. By complementing automated analysis with thorough manual verification, we achieved a deeper understanding of the real dependencies between modules.

The Importance of Synchronizing Architectural Documentation with Actual Code

Our analysis showed that as the project evolved, differences emerged between the conceptual architecture and the actual codebase. This finding highlights the necessity of regularly updating and maintaining architectural documentation to support developers' understanding and to inform future architectural evolutions.

The Complexity of Cross-Module Dependencies

We observed that certain modules exhibited multi-layered and cross-layer dependencies, which were more intricate than initially anticipated. This complexity calls for a closer examination of module coupling and suggests that potential refactoring or further abstraction could enhance overall modularity.

7. Conclusion and limitation

Obtaining the concrete architecture of GNUstep was a complex and iterative process. We discovered significant distinctions between the original conceptual model and the real structure of the system by carefully examining and considering the source. One of the primary issues was the unclear documentation, which made it hard to monitor component interconnections and dependencies. In order to minimize unnecessary complexity and enhance maintainability, this emphasizes the necessity of improved documentation procedures in open source projects.

However, our investigation was subject to several limitations. First, the Understand tool, although invaluable, occasionally generated ambiguous or incomplete dependency mappings that required additional manual scrutiny. Second, the lack of comprehensive documentation within the GNUstep codebase posed challenges in precisely interpreting some module relationships. Finally, given the evolving nature of the code, there remains the possibility that some dependencies may shift over time, potentially impacting the reproducibility of our analysis.

Overall, this study deepens our understanding of GNUstep's concrete architecture and underscores the importance of iterative analysis and up-to-date documentation. Future work should aim to refine the dependency mapping process with enhanced tooling and further validate the architectural model as the system continues to evolve.

8. Dictionary

Architecture & Software Concepts

Architecture: The organizational structure of a software system.
System: A collection of components that work together to solve a problem.
Subsystem: A modular system within a larger software system.
Environment: The runtime state in which a system operates.
Layered Architecture: A design approach that separates a system into hierarchical layers to enhance modularity and portability.
Object-Oriented Architecture (OO): A software design paradigm emphasizing class hierarchies, inheritance, and polymorphism to enhance maintainability and scalability.
Conceptual Architecture: A high-level system representation based on logical design and documentation.
Concrete Architecture: The actual system structure as observed in the codebase.

Analysis & Dependencies

Reflexion Analysis: Comparing conceptual and real architecture.
Dependency Graph: A visual map of system connections.
Absence: A missing connection in the real system.
Divergence: An unexpected connection in the real system.

9. Reference

1. GNUstep Project. (n.d.). Main Page. GNUstep MediaWiki. Retrieved March 14, 2025, from https://mediawiki.gnustep.org/index.php/Main_Page
2. Botto, F., Frith-Macdonald, R., Pero, N., & Robert, A. (2004). GNUstep manual. Free Software Foundation. Retrieved March 14, 2025, from http://andrewd.ces.clemson.edu/courses/cpsc102/notes/GNUStep-manual.pdf
3. GNUstep. (2025, January 22). In Wikipedia. Retrieved March 14, 2025, from https://en.wikipedia.org/wiki/GNUstep
4. GNUstep. (2021). README. In libs-base. Retrieved March 14, 2025, from https://github.com/gnustep/libs-base?tab=readme-ov-file