ATMACA Protocol Development Technical Diagram

Deliverable ID:	**D3.2**
Project Acronym:	**ATMACA**
Grant:	**101167070**
Call:	**HORIZON-SESAR-2023-DES-ER-02**
Topic:	**HORIZON-SESAR-2023-DES-ER2-WA2-2**
Consortium Coordinator:	**Eskişehir Teknik Üniversitesi**
Edition date:	**29 August 2025**
Edition:	**01.04**
Status:	**Official**
Classification:	**SEN**

Abstract

This document outlines the implementation of the ATMACA protocol, a next-generation communication protocol designed for evolving air traffic services. Developed within the SESAR framework, the ATMACA protocol provides a robust middleware platform that enables dynamic, secure, and standards-compliant data exchange across both airborne and ground-based systems. It introduces a modular, software-defined architecture that supports session continuity, mobility, and role-based context awareness across diverse access technologies, including SATCOM, AeroMACS, and VHF Data Link Mode 2 (VDLm2). The protocol uses a flexible messaging framework based on Datalink Information eXchange (DIX) and integrates key services such as context, session, connection, and mobility management. Designed to enable aeronautical functions such as Controller-Pilot Data Link Communications (CPDLC) and Datalink Flight Information Service (DFIS), the ATMACA protocol aligns with ICAO, EUROCAE, and SESAR interoperability objectives. Its architecture also supports emerging operational concepts such as Virtual Centres and Dynamic Airspace Configuration, offering a scalable foundation for modern Air Traffic Management (ATM) environments.

Authoring & Approval

Author(s) of the document

Organisation name	Date
Sergun Özmen / THY	20/08/2025
Hassna Louadah / DMU	20/08/2025
Raouf Hamzaoui / DMU	20/08/2025
Feng Chen / DMU	20/08/2025

Reviewed by

Organisation name	Date
Guillermo Martin Vicente / UPM	24/07/2025
Raquel Delgado-Aguilera Jurado / UPM	24/07/2025
Enes Özçelik / ESTU	28/07/2025

Approved for submission to the SESAR 3 JU by[1]

Organisation name	Date
Fulya Aybek Çetek / ESTU	28/08/2025
Stefano Bonelli and Tommaso Vendruscolo / DBL	28/08/2025
Raouf Hamzaoui / DMU	28/08/2025
Georges Mykoniatis / ENAC	28/08/2025
Matthew Cornwall / SAERCO	28/08/2025
Sergun Özmen / THY	28/08/2025
Rosa María Arnaldo / UPM	28/08/2025

Rejected by[2]

Organisation Name	Date

[1] Representatives of the beneficiaries involved in the project
[2] Representatives of the beneficiaries involved in the project

Document History

Edition	Date	Status	Company Author	Justification
01.01	10/05/2025	Draft	THY/DMU	First draft
01.02	15/07/2025	Draft	THY/DMU	Submitted for internal review
01.03	20/08/2025	Draft	THY/DMU	Revised after internal review
01.04	29/08/2025	Final	ESTU	Approved by the Consortium.

Copyright Statement

Disclaimer

ATMACA

AIR TRAFFIC MANAGEMENT AND COMMUNICATION OVER ATN/IPS

ATMACA

This document is part of a project that has received funding from the SESAR 3 Joint Undertaking under grant agreement No 101167070 under European Union's Horizon Europe research and innovation programme.

The UK participant (De Montfort University) is supported by UKRI grant number 10121610.

Table of Contents

List of Figures

List of Tables

List of Listings

Acronym	Name
AAA	Authentication, Authorisation, and Accounting
ABNF	Augmented Backus–Naur form
ACC	Area Control Center
ACM	ATC Communication Management
ACL	ATC Clearance
AeroMACS	Aeronautical Mobile Airport Communications System
AOC	Airline Operational Control
API	Application Programming Interface
APP	APPproach control unit
ATC	Air Traffic Control
ATCo	Air Traffic Controller
ATM	Air Traffic Management
ATMACA	Air Traffic Management and Communication over ATN/IPS
ATN	Aeronautical Telecommunication Network
ATSU	Air Traffic Services Unit
AVP	Attribute-Value Pairs
CM	Context Management
CMA	Context Management Application
CPDLC	Controller-Pilot Data Link Communications
DAC	Dynamic Airspace Configuration
DFIS	Digital Flight Information Service
DIX	Data Link Information eXchange
DLC	Data Link Communication
DLIC	Data Link Initiation Capability
EFB	Electronic Flight Bag
EUROCAE	European Organisation for Civil Aviation Equipment
EUROCONTROL	European Organisation for the Safety of Air Navigation
FCA	Flight-Centric Airspace
FCDI	Future Connectivity and Digital Infrastructure
FIR	Flight Information Region
FL	Flight Level
FSM	Finite-State Machine
GB-LISP	Ground-based Locator/ID Separation Protocol

GRO	Green Route Operations
HMI	Human Machine Interface
ICAO	International Civil Aviation Organisation
IP	Internet Protocol
IPS	Internet Protocol Suite
NOTAM	Notice to Airmen
OCC	Operations Control Centers
PMIPv6	Proxy Mobile IP version 6
QoS	Quality of Service
RBAC	Role-Based Access Control
RFT	Routing and Forwarding Table
SATCOM	Satellite Communications
SCTP	Stream Control Transmission Protocol
SESAR	Single European Sky ATM Research Programme
SIP	Session Initiation Protocol
SJU	SESAR Joint Undertaking
SWIM	System-Wide Information Management
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLV	Tag-Length-Value
TMA	Terminal Manoeuvring Area
TWR	ToWeR
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VC	Virtual Centre
VDLm2	VHF Data Link Mode 2
VoIP	Voice over IP

1 INTRODUCTION

1.1 Background

The increasing complexity of global aviation operations along with the growing demand for more efficient, scalable, and secure communication systems are driving a fundamental transformation in Air Traffic Management (ATM). Traditional voice-based communication methods and tightly coupled service architectures are no longer adequate to address the operational challenges of modern aviation, particularly in high-density, performance-critical airspaces.

To address these challenges, the Single European Sky ATM Research (SESAR) programme is promoting the transition toward digital, interoperable, and service-oriented communication systems across the European airspace. Within this modernisation framework, the Air Traffic Management and Communication over ATN/IPS (ATMACA) project introduces a novel protocol and system architecture designed to support ATM operations over the Aeronautical Telecommunications Network using the Internet Protocol Suite (ATN/IPS).

ATMACA provides a flexible middleware platform that integrates core communication services, including session, connection, mobility, and context management. These capabilities are designed to support a wide range of advanced aeronautical applications such as Controller-Pilot Data Link Communications (CPDLC), Digital Flight Information Service (DFIS), Green Route Operations (GRO), and System-Wide Information Management (SWIM)-enabled services while maintaining high performance, continuity, and interoperability.

Built on a modular, software-defined network architecture, the ATMACA protocol enables seamless communication between mobile and stationary clients, including aircraft avionics and Air Traffic Control (ATC) workstations. It supports persistent communication sessions, real-time context tracking, and dynamic node provisioning which are key enablers for implementing emerging SESAR concepts such as Virtual Centres (VCs), Flight-Centric Airspace (FCA), and Dynamic Airspace Configuration (DAC).

The ATMACA protocol ensures service continuity across diverse access technologies (e.g., SATCOM, AeroMACS, VDLm2), reduces pilot and controller workload, and enables automated, resilient communication management. By introducing session-based communication models and integrated message routing, the ATMACA protocol enhances safety, efficiency, and scalability in line with SESAR, International Civil Aviation Organisation (ICAO), and European Organisation for Civil Aviation Equipment (EUROCAE) requirements for next-generation ATM systems.

1.2 Purpose

This document defines the technical, functional, and architectural foundations of the ATMACA protocol. Its objectives are as follows:

- To specify the structure, behaviour, and operation of the ATMACA communication protocol, including its message formats, transport mechanisms, and key control functions.
- To offer guidance for system architects, developers, and integrators in aligning deployments of the protocol with SESAR, ICAO, and EUROCAE guidelines.
- To contribute to the broader SESAR objective of establishing a harmonised, IP-based communication infrastructure for future European ATM systems.

The document is intended for use by stakeholders involved in the design, development, integration, validation, and deployment of ATMACA-based systems, including Air Navigation Service Providers (ANSPs), industry partners, and research entities.

1.3 Scope

This document presents the functional, technical, and operational scope of the ATMACA protocol. It provides a comprehensive specification of the core services, communication mechanisms, and architectural principles that define the protocol's role in supporting secure and resilient data exchange across air and ground domains.

The scope of this document includes:

- **Protocol Specification:** Defines the ATMACA communication protocol, including message structure, session handling, error management, and Datalink Information eXchange (DIX)-based information exchange mechanisms.
- **Architecture:** Describes the protocol's modular architecture, featuring a software-defined node model, agent-based routing, and support for centralised or distributed deployments.
- **Communication Services:** Covers the protocol's foundational services, including connection management, session management, context tracking, and mobility support.
- **Service Integration:** Explains how aeronautical applications, such as Data Link Initiation Capability (DLIC), are supported and integrated within the ATMACA framework.

This document does not specify the internal implementation details of ATMACA components but provides sufficient information to support system integration, protocol compliance, and architectural design decisions across diverse operational environments.

1.4 Methodology

The development of the ATMACA protocol followed a structured and iterative engineering methodology aligned with SESAR design principles and system engineering best practices. This methodology ensures that the protocol is functionally sound, technically robust, and operationally aligned with the evolving needs of ATM environments. The methodology included the following main phases:

1.4.1 Requirement Analysis

- Identify functional and non-functional requirements specific to aeronautical environments, including session management, connection management, mobility management, and multilink support.
- Assess existing protocols, adopting relevant features while designing an independent protocol adapted to the unique demands of aeronautical communication.
- Conduct consultations with ATM professionals, service providers, pilots, and other aviation stakeholders to align the protocol design with operational, regulatory, and technological needs.
- Develop use cases deriving specifications from real-world ATM operations, airspace mobility demands, and SWIM-based data exchange scenarios.

1.4.2 Architectural Design

- Define a modular architecture that separates key functionalities, such as session management, mobility management, and connection management, to allow flexibility and scalability.
- Achieve scalability through software-defined nodes, where dedicated agents are deployed across the network in predefined locations. When an aircraft enters a specific region, the assigned agent dynamically manages its sessions, ensuring efficient resource allocation, optimised network load distribution, and seamless service continuity as the aircraft transitions between regions.
- Develop a DataLink Context Management (DLCM) application to coordinate session management (establishing and maintaining communication sessions), connection management (ensuring secure and reliable links over ATN/IPS), and mobility management (facilitating smooth handovers between network agents as aircraft transition across regions). DLCM plays a critical role in ensuring seamless, scalable, and resilient air-ground communication, optimising network resource allocation, and enhancing operational efficiency within ATN/IPS-enabled aeronautical environments.
- Develop a robust signalling mechanism inspired by SIP [8] for session control and the Diameter Base Protocol [5] for authentication and security while ensuring a unique approach adapted to the specific requirements of aeronautical communication.
- Integrate multilink support to enable dynamic selection, aggregation, and failover mechanisms.

1.4.3 Protocol Specification Development

- Develop detailed specifications for each protocol component, including message formats, signalling flows, and state machine behaviour.
- Define mobility management mechanisms, such as handover types (soft/hard, vertical/horizontal, intra/inter), to address seamless transitions.
- Specify security protocols (e.g., Transport Layer Security (TLS) 1.3) for encrypted communication and robust authentication.

Throughout these phases, stakeholder collaboration is maintained to validate assumptions, refine design decisions, and ensure that the ATMACA protocol meets the needs of both operational users and system developers. This methodology enables continuous feedback, traceability, and adaptability throughout the protocol's lifecycle.

1.5 Structure of the Document

The remainder of this document is organised as follows. Section 2 provides an overview of the ATMACA protocol architecture, including its modular structure, core components, and communication model. Section 3 presents implementation details and offers guidance for application developers, covering aspects such as message formats, session handling, and integration with aeronautical services. Section 4 explains the architectural composition of the ATMACA software stack, describing the primary modules, their runtime interactions, class responsibilities, and deployment models. Finally, Section 5 summarises the key points and presents future steps.

2 ATMACA PROTOCOL - ARCHITECTURE OVERVIEW

ATMACA is a modern framework-based solution designed to enhance ATM by enabling secure, reliable, and efficient data exchange between airborne and ground-based systems. Built on an IP-based infrastructure, ATMACA exploits the flexibility and scalability of the Internet Protocol to support seamless connectivity and efficient data transmission across diverse communication networks. By addressing key challenges in traditional communication systems, such as session continuity, mobility management, and security, ATMACA provides a comprehensive set of features and capabilities that enhance network performance, service availability, and operational resilience.

2.1 ATMACA Solution Overview

As a framework-based solution, ATMACA offers a flexible and extensible platform that simplifies the integration of communication services into ATM applications. By providing core services such as connection, context, session, and mobility management, ATMACA enables developers to focus on application-specific functionalities rather than implementing low-level communication infrastructure (**Figure 2.1**). This approach accelerates the development cycle, reduces complexity, and promotes faster deployment of new ATM solutions.

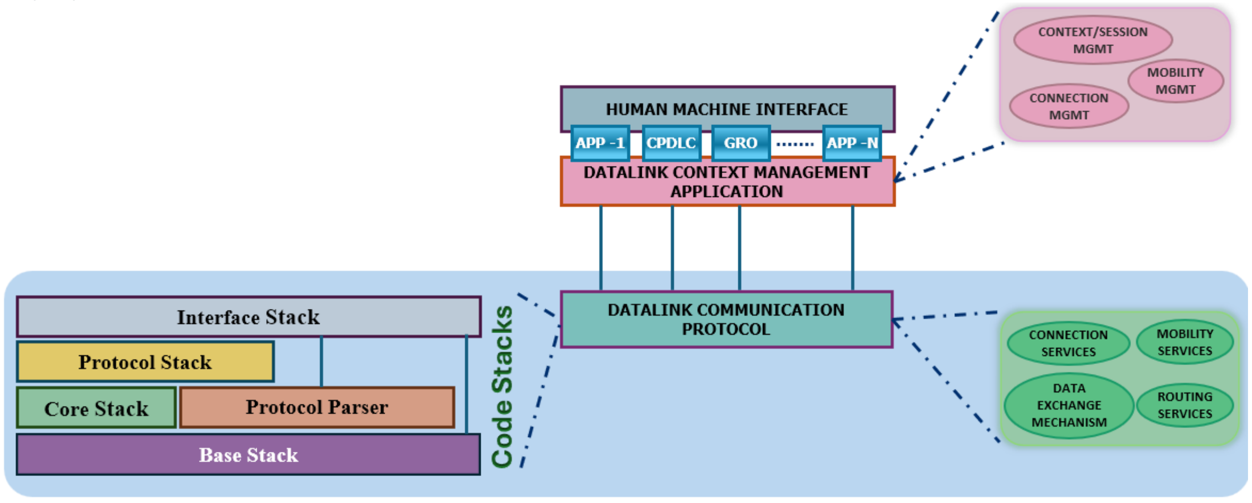


Figure 2.1: ATMACA Solution Conceptual Architecture.

ATMACA integrates advanced capabilities like multilink support for redundancy, load balancing strategies, and dynamic handover mechanisms to maintain uninterrupted connectivity during aircraft movement. Enhanced mobility management ensures seamless transitions between different communication technologies such as VDL, AeroMACS, and satellite links, minimising packet loss and ensuring continuous communication across airspace boundaries. To enhance reliability, ATMACA incorporates fault-tolerance mechanisms, including high-availability clustering, N+1 redundancy, and dynamic failover strategies, significantly reducing the risk of service disruptions.

The ATMACA protocol also uses software-defined nodes, which provide enhanced flexibility, scalability, and adaptability by decoupling hardware from network control functions. These nodes enable dynamic reconfiguration, allowing the system to respond to network changes, prioritise critical services, and optimise resource allocation in real-time. This approach enhances performance by enabling adaptive routing strategies and automated fault recovery, ensuring network stability even during peak traffic conditions.

ATMACA’s comprehensive connection, context and session management capabilities provide a solid foundation for ensuring stable communication between nodes and services. It efficiently manages session initiation, maintenance, and termination while mitigating packet loss during network disruptions and transitions. The protocol mobility management extends across all key aspects including user, session, service, and terminal mobility ensuring communication continuity as users, services, or terminals move between network domains. These features are critical for maintaining seamless air traffic control services, particularly in high-mobility environments like aviation.

ATMACA’s flexible architecture effectively supports both centralised and distributed deployments, enhancing system resilience through redundancy strategies and efficient resource management. The ATMACA solution seamlessly integrates with essential air traffic services such as DFIS and CPDLC, enhancing operational efficiency and reducing the risk of miscommunication. Its modular design ensures adaptability to future advancements, supporting scalability to meet the growing demands of modern air traffic operations. Furthermore, the ATMACA solution includes intuitive human-machine interfaces to improve usability, reducing controller and pilot workload while enhancing situational awareness. By aligning with ICAO's NextGen and SESAR initiatives, ATMACA ensures global compliance, positioning itself as a versatile, adaptable, and forward-looking solution for efficient and scalable air traffic management.

2.1.1 Datalink Communication Protocol

The Datalink Communication Protocol, referred to as the base protocol in this document, forms the core communication layer of the ATMACA protocol. It provides essential networking functions such as data exchange mechanisms, connection, mobility, and routing services (**Figure 2.1**). It ensures the secure and reliable transmission of data across different network layers, serving as the backbone for communication within the ATMACA system. The base protocol maintains stable connections and enables efficient data flow, supporting robust communication for air traffic services.

A key feature of the ATMACA Datalink Communication Protocol is its integration of software-defined nodes, which enhances flexibility, scalability, and adaptability. Unlike traditional network nodes that rely on fixed hardware configurations, software-defined nodes decouple hardware from control functions, allowing greater agility in managing network resources and using alternative communication paths. This design empowers ATMACA to dynamically adjust routing decisions, allocate bandwidth, and prioritise critical services based on real-time network conditions.

Communication between software-defined nodes is established through two primary methods:

- **Peer-to-Peer Connections:** In this model, nodes communicate directly via dedicated physical links, ensuring fast and reliable data exchange. This method is particularly useful in scenarios requiring low latency and high data throughput, such as communication between critical ATC units or control centres.
- **Session-based Connections:** In addition to physical links, ATMACA supports logical connections that are established via managed sessions. These session-based connections provide flexibility by enabling communication between nodes without requiring awareness of the underlying physical links. Logical connections allow nodes to establish virtual communication channels that dynamically adapt to changing network conditions. This approach is essential for ensuring session continuity and uninterrupted communication as aircraft or network nodes transition between different access points.

By combining peer-to-peer connections with session-based connections, ATMACA ensures both robust data delivery and flexible communication paths. This hybrid approach enhances resilience by allowing the system to intelligently switch between communication modes and paths in response to network congestion, link failures, or changing operational priorities.

2.1.2 Foundational Application

The Datalink Context Management Application builds on the base protocol and extends its capabilities by providing key services such as context, session, mobility, and connection management (**Figure 2.1**). It is the foundational application layer of the ATMACA protocol, serving as a runtime coordination environment that bridges aeronautical applications such as CPDLC, DFIS, and GRO with the underlying communication infrastructure. It enables context-aware, role-based, and mobility-resilient service operations across both stationary and mobile nodes. Evolving from the traditional DLIC of the ATN/OSI model, the DLCM in ATN/IPS introduces a broader, service-oriented architecture capable of sustaining seamless operation under dynamic network and operational conditions.

DLCM delivers an integrated set of services critical for distributed air traffic communications. These services include:

- **Connection Management:** Manages the establishment, maintenance, and termination of communication links between devices and/or nodes. It enables reliable data transfer through link quality monitoring, bandwidth allocation, and recovery mechanisms implementation addressing connection failures or disruptions problem.
- **Context Management:** This feature dynamically adapts communication protocols and services based on real-time contextual information, such as environmental conditions, network performance, or mission-critical priorities. By intelligently adjusting communication parameters, Context Management optimises data flow, enhances service quality, and optimises resource utilisation in dynamic ATM environments.
- **Session Management:** Ensures stable communication between nodes by effectively managing session initiation, maintenance, and termination, while mitigating packet loss during network transitions or disruptions.
- **Mobility Management:** Provides comprehensive support across four key dimensions:
    - **User Mobility:** Ensures that users retain a consistent identity and access personalised services across various devices or locations.
    - **Session Mobility:** Enables active communication sessions to continue uninterrupted as a user switches between devices or as a device moves across different networks.
    - **Service Mobility:** Ensures that users can access the same application services across different networks and devices.
    - **Terminal Mobility:** Allows a mobile client to maintain continuous network connectivity regardless of changes in its physical location.

2.1.3 Human Machine Interface

The Human Machine Interface (HMI) serves as the primary interface for end-users, such as Air Traffic Controllers (ATCos) and flight crew, to interact with the ATMACA communication system. Designed for usability and efficiency, the HMI offers intuitive displays, streamlined controls, and enhanced situational awareness tools. By simplifying complex data presentation and improving user interaction, the HMI reduces workload for controllers and pilots, enhancing safety and operational efficiency.

The HMI integrates key ATM services such as DFIS and CPDLC. These services are seamlessly accessible through the interface, ensuring smooth and effective communication during critical airspace operations.

2.2 ATMACA Software Defined Nodes

The ATMACA solution introduces a sophisticated communication framework designed to support the high-mobility and dynamic demands of aeronautical environments. At its core lies a structured network of software-defined nodes that enable seamless communication, efficient mobility management, and reliable data exchange between nodes (clients, servers, and agents). The ATMACA protocol uses a modular architecture to ensure scalable and extensible communication in modern air traffic networks.

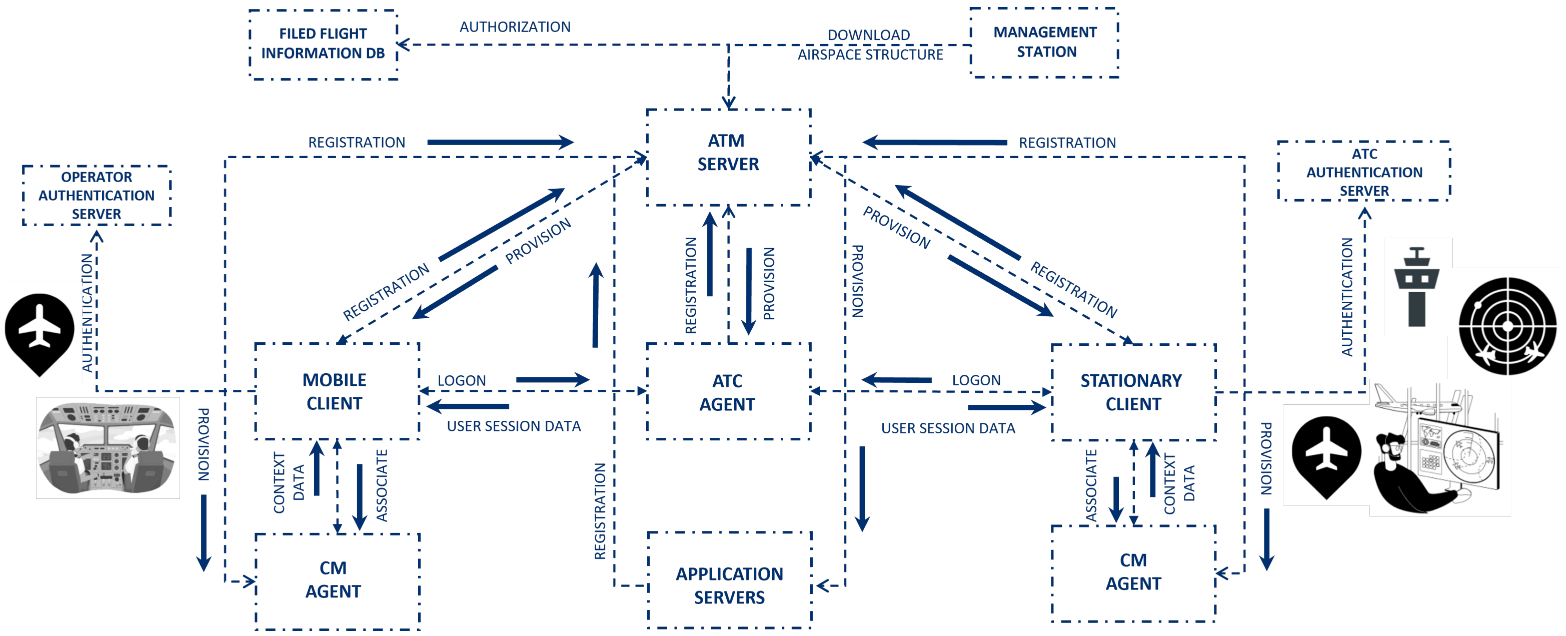


Figure 2.2: ATMACA's Network Architecture.

The ATMACA network architecture uses a layered structure where software-defined nodes are key elements and responsible for managing communication and operational tasks. These nodes include ATM Server, Agents (ATC and CM), Clients (mobile and fixed), and external systems dealing with the registration, authentication, and provisioning of users and session data across the network (**Figure 2.2**)

Each of these components plays a vital role in ensuring data integrity, seamless session management, and operational efficiency. The ATMACA software-defined node architecture optimises communication by dynamically adjusting node provisioning, enabling efficient traffic routing, and ensuring uninterrupted services in diverse aeronautical conditions.

Figure 2.2 illustrates the ATMACA network architecture, highlighting the interaction between software-defined nodes. ATM servers provision the nodes, while ATC agents manage efficient data routing using context data provided by Context Management (CM) agents, enabling communication between authenticated clients. Application Servers are responsible for delivering essential services.

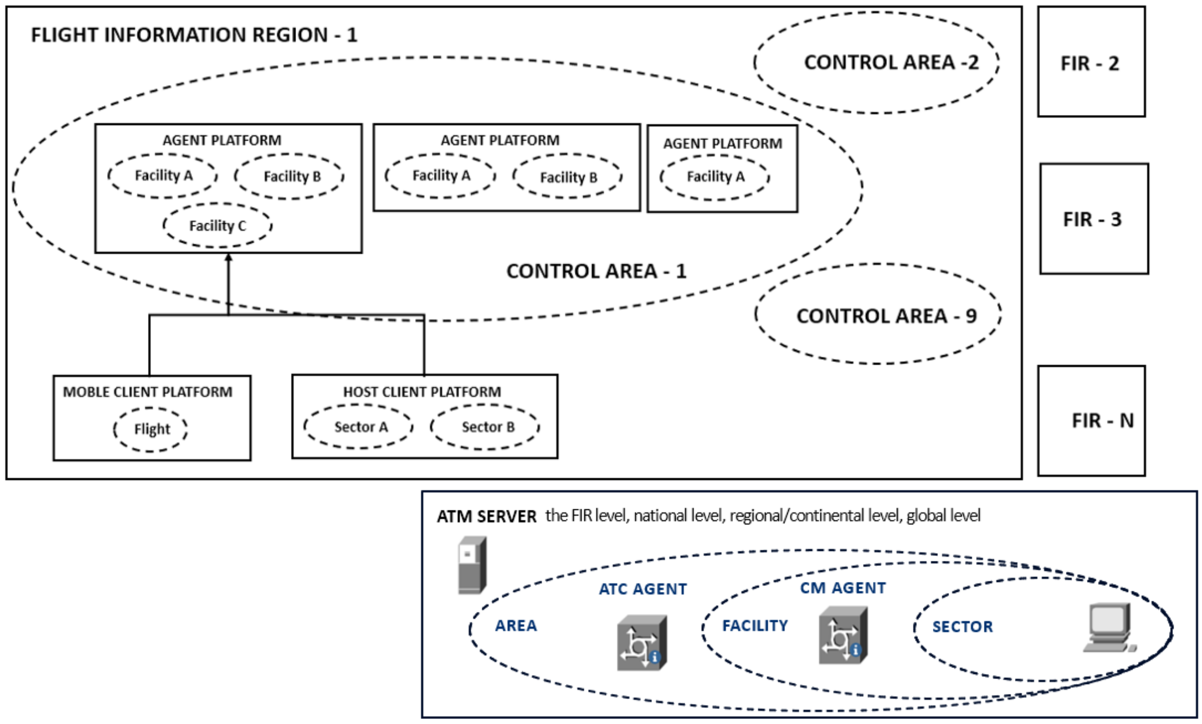


Figure 2.3: ATMACA Airspace Structure and Logical Architecture.

The ATMACA airspace architecture supports flexible network designs by organising operational domains hierarchically. **Figure 2.3** depicts a layered deployment of the ATMACA network nodes across multiple Flight Information Regions (FIRs), Control Areas, facilities, and sectors. It highlights how agents, client platforms, and services are logically organised within each region. FIRs contain multiple Control Areas, which include various facilities covering one or more sectors managed by agents. Each agent platform supports mobile and host clients, such as aircraft systems or ATC workstations.

ATM Servers act as provisioning and policy anchors within the system. Depending on the deployment strategy, an ATM Server can operate at an FIR, national, regional/continental, or even at a global level. This design flexibility allows air ANSPs to align ATMACA deployment with their operational, geographical, and regulatory constraints.

The ATC and CM Agents operate at the facility level, maintaining real-time awareness of connected clients, managing communication handovers, and ensuring seamless session continuity across network transitions or disruptions. This distributed model enhances resilience, ensures scalability, and supports cross-border coordination in line with SESAR objectives.

2.2.1 Servers

In the ATMACA network, servers are fundamental components responsible for delivering centralised services, enforcing policies, and maintaining the system’s operational integrity. These servers are part of the software-defined node architecture and serve as coordination points for communication, service provisioning, authentication, and data management across the network (**Figure 2.2**).

ATMACA defines multiple server roles, including the ATM Server, which manages system configuration, node registration, and airspace provisioning; and the Application Server, which hosts and delivers operational services such as DFIS and weather distribution.

Servers' ability to function in both centralised and distributed deployments, allows ATMACA to adapt to a wide range of air traffic management scenarios and stakeholder needs.

2.2.1.1 ATM SERVER

The ATM SERVER software runs on a server platform and is responsible for provisioning, authorisation, and security services across the network nodes. Depending on network design considerations and strategies, the ATM SERVER can be configured in either a centralised or distributed architecture.

In a centralised architecture setup, a single ATM SERVER manages all network operations where redundancy is essential to eliminate single points of failure and enhance system reliability. This can be achieved using a high-availability clustering, where multiple interconnected servers monitor each other's status. If one server fails, another takes over its responsibilities seamlessly, minimising downtime. Another method is N+1 redundancy, which involves deploying a standby server alongside several active servers. The standby server remains idle during normal operations and activates only if an active server fails. This approach offers a cost-effective solution for maintaining fault tolerance.

In a distributed architecture, server responsibilities are spread across multiple interconnected servers. This setup improves both system reliability and scalability. It eliminates single points of failure and allows for more efficient handling of incoming requests. Load balancing plays a key role in this architecture by distributing traffic among servers. Techniques such as round-robin or advanced scheduling algorithms help ensure no single server becomes overloaded. As a result, performance and response times are optimised.

By adopting a centralised architecture with redundancy strategies or a distributed architecture with effective load balancing, the ATMACA system can achieve high availability and efficient resource utilisation, ensuring robust and reliable network operations.

The ATM Server maintains a comprehensive and authoritative model of the operational airspace, encompassing detailed definitions of sectors, control zones, FIRs, and their associated vertical and horizontal dimensions. It also tracks the relationships between adjacent sectors and areas to support inter-unit coordination. To populate and update this structure, the Management Station acts as the primary configuration authority by downloading validated definitions from the administrator and pushing them to the ATM Server as needed (**Figure 2.2**). This includes full datasets for sector, area, and facility tables, which together define the organisational and geographic layout of the controlled airspace (**Figure** 2.3).

During the synchronisation process, a filtered version of the node management table (containing only relevant node attributes such as Node ID, Type, Role, and associated sector/area/facility mappings) is used. This subset allows the ATM Server to maintain a real-time Node-Role mapping cache which is essential for routing messages, assigning context control, and enforcing communication policies. By separating topology from performance data, the system ensures that the ATM Server operates with a stable, accurate view of the airspace and network environment, while supervisory and diagnostic tasks remain within the Management Station. The Management Station is considered external to the ATMACA software-defined node architecture.

The ATM SERVER maintains comprehensive knowledge of the airspace structure (**Figure 2.3**), including:

EUROPEAN PARTNERSHIP

Co-funded by the European Union

- **Sectors**: Defined portions of airspace managed by specific ATC units.
- **Control Zones**: A controlled airspace that normally extends from the surface to a specified upper limit around an aerodrome. Its purpose is to protect aircraft during arrival and departure phases [11].
- **Flight Information Regions**: Large airspace regions with specific boundaries where flight information and alerting services are provided.
- **Vertical and Horizontal Dimensions**: Altitude layers and lateral boundaries defining the extent of airspace segments.
- **Adjacent Sectors or Areas**: Neighbouring airspace sections requiring coordination to ensure seamless traffic management.

The ATM SERVER provides each network node with relevant airspace information upon request, ensuring that all nodes receive accurate and up-to-date data. A designated authority, such as an ANSP or as defined by ICAO regulations, is responsible for maintaining data integrity and defining the airspace structure and provisioning rules. These configurations are uploaded and managed by the ATM SERVER to ensure compliance with established protocols. During registration, nodes such as agents and endpoints are assigned to specific airspace sectors or domains, based on the configuration provided by the air traffic authority, in line with predefined structures like FIRs, sectors, or zones. The ATM SERVER also supports dynamic node provisioning in response to changes in airspace, traffic conditions, or operational needs. This capability enables efficient routing, conflict detection, and air traffic management. To maintain consistency and reliability, the ATMACA system applies redundancy strategies, ensuring secure, synchronised, and up-to-date airspace information across all nodes in the network.

2.2.1.2 APPLICATION SERVER

The ATMACA Application Server is a specialised platform designed to host, manage, and deliver a comprehensive range of aeronautical services, ensuring accessibility for both stationary and mobile clients such as pilots, air traffic controllers, and dispatch teams (**Figure 2.2**). Applications are mainly running on client devices (such as ATC workstations or aircraft systems) while services are hosted and delivered by the Application Server. The Application Server plays a pivotal role in enabling efficient and scalable air traffic management operations through its distributed architecture and integration with the ATMACA Agents.

To avoid confusion, ATMACA distinguishes between the terms application and service as follows:

- **Application:** User-facing software components residing on client devices (e.g., pilot Electronic Flight Bag (EFB) or ATC console) that interact with ATMACA services.
- **Service:** Backend logic hosted on the Application Server, which provides data, processing, and support to applications across the network.

Some applications may be hybrid, with part of their logic running locally, and the other part is served remotely from the Application Server (e.g., session state tracking, mobility context updates). The ATMACA Application Server supports a variety of essential services, including DFIS, GRO, weather data distribution, and ATM tools and services.

The Application Server's distributed architecture provides high availability, scalability, and efficient load balancing by replicating server instances across multiple nodes and dynamically allocating resources to adapt to network demands. Intelligent routing prevents server overload, while strategic deployment across geographic and operational domains ensures low-latency, localised service delivery to users such as pilots and controllers.

The ATMACA Application Server delivers transaction-based and session-based services. Transaction-based services are provided as discrete, stateless interactions decoupled from previous requests (e.g., flight information requests, ATC clearance requests). Session-based services are provided as stateful interactions that maintain communication context throughout an ongoing session (e.g., CPDLC, monitoring services, and Voice over IP (VoIP) calls).

The ATMACA Application Server can be integrated with the SWIM infrastructure. Through standardised APIs and protocols, the Application Server enables secure and efficient data exchange with external systems, including regional SWIM networks. This integration ensures smooth communication with global air traffic management frameworks and enhances overall service coordination.

2.2.2 Agents

An AGENT in the ATMACA network is a software component and an intermediary network node responsible for facilitating communication, coordination, and data management between network entities. AGENTS play a key role in ensuring seamless interaction between clients and other network nodes by maintaining peer awareness, processing context-related messages, managing connections, and supporting mobility (**Figure 2.2**).

Operating as intelligent intermediaries, AGENTS are designed to route requests to users' current location based on defined routing policies and uploaded location data, ensuring transparent and efficient message delivery for both mobile and stationary nodes. They dynamically adapt to network changes to provide consistent and reliable communication pathways.

In addition to routing, AGENTS manage key operations such as role transitions, context registration, metadata synchronisation, and status tracking, ensuring service continuity and operational stability. By maintaining synchronised and accessible context data across the network, AGENTS enhance scalability, fault tolerance, and system resilience.

The ATMACA protocol defines two specialised AGENT types with distinct responsibilities, ATC and CM agents. The ATC Agent plays a central role in supporting operational data flow within the ATMACA network. It provides forwarding and routing services for ATC communications, ensuring that messages are efficiently relayed between air traffic control systems and aircraft. In addition, it facilitates the reliable delivery of these messages across dynamic network environments and manages essential communication lifecycle functions such as logon, logout, and registration. These functions are critical for maintaining stable and continuous communication between airborne and ground-based systems.

The CM Agent is responsible for organising, maintaining, and synchronising context-related data within the ATMACA network. It manages associations between workstations and operational roles, assigning roles such as controlling, mirroring, and monitoring to ensure coordinated service delivery. The CM Agent also handles the full lifecycle of context-related operations, including role transitions (e.g., handover and takeover), context uploads and downloads, and real-time status queries. Through these capabilities, the CM Agent ensures operational continuity, accurate metadata synchronisation, and seamless communication handovers between nodes.

Looking ahead, the ATMACA solution is designed to support the introduction of additional agent types as new operational requirements and services emerge. These may include agents dedicated to specialised domains such as airspace configuration management, collaborative decision-making, SWIM gateway interfacing, or trajectory-based operations [12]. The agent model's extensibility ensures that the architecture can adapt to evolving ATM practices while maintaining consistency in routing, context handling, and service mediation across all network participants.

2.2.3 Clients

In the ATMACA system, clients are software components that interact with the network to exchange messages, request services, and maintain communication sessions (**Figure 2.2**). These clients are deployed on physical platforms such as aircraft systems, air traffic control workstations, or portable mobile devices. Collectively, the client software and its associated hardware form a Mobile or Stationary Node, depending on the operational context.

ATMACA clients connect to the network through agents and servers to enable data link services, participate in session and context management, and support mobility transitions during dynamic flight or control operations.

ATMACA defines two primary categories of clients, based on their deployment and operational behaviour, Mobile and Stationary clients. A Stationary Client maintains a fixed association with a single ATC Agent, meaning its point of attachment does not change throughout its operational lifecycle. While the physical location of a stationary client (e.g., a vehicle or handheld terminal) may move within a facility or airport, it is still considered stationary from a protocol perspective because it consistently communicates through the same ATC Agent. In contrast, a Mobile Client is defined by its ability to transition between different ATC Agents as it moves across control areas or FIR boundaries. This dynamic attachment capability requires additional mechanisms for session and context mobility, allowing seamless handovers between agents without service disruption. The distinction is not about physical mobility, but rather about ATC Agent anchoring which forms the basis for how mobility management and rebinding logic are applied within the protocol stack.

Stationary Clients maintain a connection to their assigned local ATC Agent and ATM Server within a single network domain and do not initiate agent handovers. Instead, they provide operational functions such as issuing clearances, managing CPDLC sessions, and monitoring user or client mobility events across the network.

Mobility management mechanisms continuously update the network with the client's location, context status, and session bindings, enabling seamless rerouting, rebinding, and service continuity. Mobile clients are optimised for real-time applications such as CPDLC exchanges, flight plan updates, weather requests, and airborne alerts ensuring persistent connectivity regardless of their geographic or network position.

3 ATMACA PROTOCOL SPECIFICATION

The ATMACA Communication Protocol integrates robust mechanisms for authentication and authorisation along with context, connection, and session management capabilities, as well as mobility support, ensuring reliable connectivity in dynamic aviation environments. The protocol operates independently of underlying transport layers, supporting Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Stream Control Transmission Protocol (SCTP) over dedicated ports for various services such as DLCM, CPDLC, and DFIS.

Incorporating a flexible, modular structure, ATMACA's message design uses DIX fields to enable efficient message formatting, error reporting, and service-specific extensions. The application-ID in each message ensures delivery to the intended aeronautical application, supporting accurate and reliable communication. Additionally, the protocol's robust error management framework, implemented through Response-Code DIX, provides clear feedback in case of failed transactions, session terminations, or unavailable services.

ATMACA uses a Session-Id DIX model that ensures seamless session continuity during aircraft mobility transitions. This identifier maintains session context across network boundaries, ensuring uninterrupted services. ATMACA's Registration, Logon, and Update functions provide smooth handovers as aircraft move between airspace sectors. To enhance reliability, ATMACA introduces periodic heartbeat messages to verify active connections and detect network failures promptly.

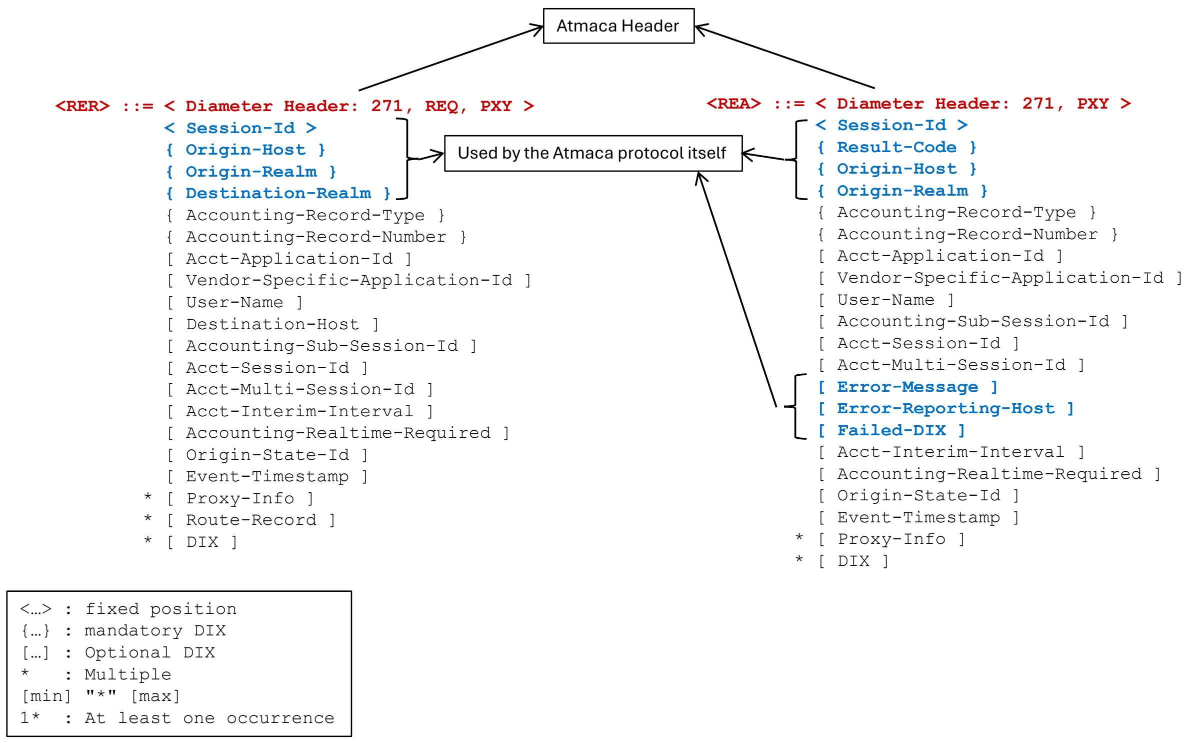
3.1 ATMACA Message Structure

The ATMACA Protocol defines a structured binary message format that ensures reliable communication for aeronautical data exchange. The protocol message structure is designed to support flexible data handling, robust session control, and extensible information exchange through DIX elements. Below is a detailed explanation of the ATMACA message format, including its Header, DIX structure, and associated data types.

3.1.1 ATMACA Message Format Overview

An ATMACA message comprises two main components: message header and DIX entities. The ATMACA message header controls message identification, structure, and routing while DIX entities contain detailed information elements for session control, mobility, and service data.

The Augmented Backus–Naur form (ABNF) structure [6] of the ATMACA messages, which is inspired by Diameter base protocol [5], is shown in **Figure 3.1**. The ATMACA messages are named with Command-Code names. Diameter command names typically include one or more English words followed by the verb "Request" or "Answer". Each English word is delimited by a hyphen. A three-letter acronym for both the request and answer is also normally provided. For example, the REgister-Request (RER) message is used to transmit accounting information to the Diameter server, which MUST reply with the REgister-Answer (REA) message to confirm reception.



```
<RER> ::= < Diameter Header: 271, REQ, PXY >
          < Session-Id >
          { Origin-Host }
          { Origin-Realm }
          { Destination-Realm }
          { Accounting-Record-Type }
          { Accounting-Record-Number }
          [ Acct-Application-Id ]
          [ Vendor-Specific-Application-Id ]
          [ User-Name ]
          [ Destination-Host ]
          [ Accounting-Sub-Session-Id ]
          [ Acct-Session-Id ]
          [ Acct-Multi-Session-Id ]
          [ Acct-Interim-Interval ]
          [ Accounting-Realtime-Required ]
          [ Origin-State-Id ]
          [ Event-Timestamp ]
        * [ Proxy-Info ]
        * [ Route-Record ]
        * [ DIX ]
```

```
<REA> ::= < Diameter Header: 271, PXY >
          < Session-Id >
          { Result-Code }
          { Origin-Host }
          { Origin-Realm }
          { Accounting-Record-Type }
          { Accounting-Record-Number }
          [ Acct-Application-Id ]
          [ Vendor-Specific-Application-Id ]
          [ User-Name ]
          [ Accounting-Sub-Session-Id ]
          [ Acct-Session-Id ]
          [ Acct-Multi-Session-Id ]
          [ Error-Message ]
          [ Error-Reporting-Host ]
          [ Failed-DIX ]
          [ Acct-Interim-Interval ]
          [ Accounting-Realtime-Required ]
          [ Origin-State-Id ]
          [ Event-Timestamp ]
        * [ Proxy-Info ]
        * [ DIX ]
```

```
<…> : fixed position
{…} : mandatory DIX
[…] : Optional DIX
*   : Multiple
[min] "*" [max]
1*  : At least one occurrence
```

Figure 3.1: ABNF Structure of ATMACA Messages.

Figure 3.1 illustrates the ABNF definitions of the ATMACA protocol's RER and Register REA messages, showing both their required and optional Attribute-Value Pairs (AVPs) for session and error handling.

3.1.2 ATMACA Header Format

The ATMACA header (**Figure 3.2**) is the foundational component of every ATMACA Protocol Data Unit (PDU). It ensures efficient message management, session control, and error handling.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Ver.|-|P|P|T|R|                Message Length                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Application-ID             |        Command-Code         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Request-ID                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  DIXes  ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.2: ATMACA Protocol Header Format.

The binary layout of the ATMACA protocol header includes fields such as Version, Flags (Priority, Retransmission, Request), Message Length, Application-ID, Command-Code, Request-ID, and DIX payload area. **Table 3.1** provides details of each field included in the ATMACA header.

Table 3.1: Header Field Description

Field	Size	Description
Version (Ver.)	3 bits	Identifies the protocol version (Current version = 0).
Priority (P)	2 bits	Determines message priority for delivery.
Retransmission Flag (T)	1 bit	Signals whether the message is a retransmission.
Request Flag (R)	1 bit	Identifies whether the message is a **request** (R=1) or **response** (R=0).
Message Length	16 bits	Specifies the total length of the ATMACA message, including the padded DIX elements. The message length is always a multiple of 4.
Application-ID	16 bits	Identifies the intended ATMACA application or vendor-specific service.
Command-Code	16 bits	Identifies the command associated with the message (e.g., Register, Logon, Service Request).
Request-ID	32 bits	Tracks session-based transactions to link requests and corresponding responses.
DIX Entries	Variable	Encapsulated information elements for data exchange.

3.1.3 DataLink Information Exchange

The ATMACA protocol uses DIX as the primary data structure to facilitate communication between nodes. Each DIX is a structured data unit that carries information about the user's session, mobility data, or control commands. DIX is designed to ensure flexibility, extensibility, and compatibility with various aeronautical communication scenarios. DIXes are embedded in ATMACA messages for: Session Control, Mobility Management, Error reporting, Service Delivery (**Figure 3.3**). DIXes have the following features:

- **Structured Format:** Each DIX carries well-defined data attributes, like Diameter's AVPs [5].
- **Flexible Encoding:** Supports various data types (e.g., integer, string, binary) to allow rich information exchange.
- **Extensibility:** New DIX types can be defined to support evolving aeronautical communication services.
- **Session Continuity:** Each DIX is tied to a Session-ID, ensuring all messages in a session are linked to the same logical flow.
- **Error Handling:** DIX entries include error codes for quick fault detection and recovery.

Each DIX consists of a structured header followed by a data field, allowing seamless integration of session control, service requests, and mobility management details (**Figure 3.3**). With dedicated flags for vendor-specific attributes, mandatory processing, and data protection, the DIX structure ensures reliable data interpretation and robust error handling.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V M P d d d d d|                 DIX Length                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          DIX Code                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Vendor-ID (opt)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Data ...    +-+-+-+-+-+-+-+-+
```

Figure 3.3: DIX Field Format.

Table 3.2 provides details of the DIX field used in the ATMACA protocol, including control flags (Vendor-Specific, Mandatory, Protected), data type bits, DIX length, DIX code, optional Vendor-ID, and the associated data payload.

Table 3.2: DIX fields.

Field	Description
V (Vendor-Specific)	Indicates if the DIX belongs to a vendor-specific space.
M (Mandatory)	Signals whether the receiver must understand this DIX.
P (Protection Flag)	Specifies the need for end-to-end security.
d-bits (Data Type)	Encodes the DIX data type (e.g., OctetString, Integer32, etc.).
DIX Length	Specifies the total length of the DIX, including padding.
DIX Code	Identifies the specific DIX attribute.
Vendor-ID (Optional)	Uniquely identifies the vendor if the V bit is set.
Data	Contains the actual payload of the DIX.

DIX supports multiple data types to ensure flexibility in data encoding (as shown in **Table 3.3**). Moreover, Grouped DIX type allows multiple individual DIX elements to be nested within a parent DIX, enabling the efficient organisation of related information under a single logical structure. In ATMACA, this concept provides a structured way to encapsulate logically related fields into reusable, modular units.

Table 3.3: DIX Data Types.

Data Type	Value
DIX_DATATYPE_OCTETSTRING	0
DIX_DATATYPE_INTEGER32	1
DIX_DATATYPE_INTEGER64	2
DIX_DATATYPE_UNSIGNEDINT32	3
DIX_DATATYPE_UNSIGNEDINT64	4
DIX_DATATYPE_FLOAT32	5
DIX_DATATYPE_FLOAT64	6
DIX_DATATYPE_GROUPED	7 (Special type for nested DIX elements)

While Grouped DIXes are directly applied to Common DIXes, such as node descriptors and peer endpoint metadata, their use is not limited to these. Any collection of DIX fields, including future protocol extensions and application-specific definitions, can be grouped under a named structure to promote consistency, modularity, and extensibility. For example, groups like NODE-DIX, ORIGIN-DIX, and DEST-DIX, help protocol messages remain clean and organised, while enabling flexible reuse across message types and functional layers.

Grouped DIXes also improve message readability, simplify parsing, and future-proof the protocol by making it easier to introduce new capabilities without disrupting existing formats. The next section introduces the set of mandatory Common DIXes defined in ATMACA, which serve as the foundational metadata used across all standard message types and are formally structured as Grouped DIXes.

3.1.3.1 Common DIXes

ATMACA protocol messages are structured using DIX elements, which represent standardised data fields exchanged between communicating nodes. A core set of Common DIXes is included across most ATMACA messages to ensure consistent identification, addressing, and session tracking. These DIXes encapsulate metadata such as context identifiers, session references, node identity and role information, and endpoint descriptors for origin and destination entities (**Table 3.4**). To support modularity and reuse, these fields are organised into logical groups (including NODE-DIX, ORIGIN-ENDPOINT, and DEST-

ENDPOINT) allowing protocol messages to remain extensible and adaptable while preserving interoperability. Building on this shared metadata layer, each ATMACA-enabled application (such as CPDLC, DFIS, or GRO) can define its own application-specific messages by extending the base structure. These custom messages inherit the standardised DIX fields, ensuring alignment with the underlying session, context, and mobility management architecture, while allowing each application to focus on service-specific logic and data. This modular approach promotes reusability, extensibility, and interoperability across diverse operational environments and service domains. These DIX elements are included in all ATMACA messages and provide the foundational identity, addressing, and routing metadata required for context and session coordination.

Table 3.4: COMMON DIXes.

Group DIX	DIX Name	Description
Base-Dix	Context-ID	Unique identifier for the context (logical communication scope shared by related peers).
	Session-ID	Unique identifier for a session between two endpoints.
	Vendor-ID	Identifier for the system or vendor generating the message.
Node-Dix	NodeName	Logical name or ID of the local node (e.g., "acft-123", "atc-east").
	NodeType	Indicates if the node is a SERVER, AGENT or CLIENT.
	NodeRole	Role of the node in the architecture: options include: ATM SERVER, ATC AGENT, CM AGENT, MOBILE CLIENT, or STATIONARY CLIENT.
	NodeRealm	Domain, region, or administrative realm of the node (e.g., FIR ID or organisational boundary).
	NodeHost	Host identifier or platform label for the node.
	NodeConnAddr	Connection address (e.g., IP:Port or URI) used to reach the node.
Origin-Dix	OrigName	Name or identifier of the **origin** node in the message.
	OrigType	Type of the origin node: must be either SERVER, AGENT or CLIENT.
	OrigRole	Role of the origin node: one of: ATM SERVER, ATC AGENT, CM AGENT, MOBILE CLIENT, STATIONARY CLIENT.
	OrigRealm	Realm or domain of the origin node.
	OrigHost	Hostname or system ID of the origin node.
	OrigConnAddr	Connection address for the origin node.
Dest-Dix	DestName	Name or identifier of the **destination** node.
	DestType	Type of the destination node: SERVER, AGENT or CLIENT.
	DestRole	Role of the destination node: ATM SERVER, ATC AGENT, CM AGENT, MOBILE CLIENT, STATIONARY CLIENT.
	DestRealm	Realm or domain of the destination node.
	DestHost	Hostname or system ID of the destination node.
	DestConnAddr	Connection address for the destination node.

3.1.3.2 Error Handling and Error-Related DIXes

ATMACA uses in-band error signalling by embedding error-related DIX fields directly within response messages (**Table 3.5**). This approach aligns with modern protocol practices, such as those found in Diameter [5], where the outcome of a request, whether successful or erroneous, is communicated using the same message type. The Result-Code DIX is used to indicate the outcome of an operation, with standardised values representing success, protocol errors, authorisation failures, or internal system issues. In cases of failure, additional DIX elements such as Error-Message, Extended-Result-Code, Failed-DIX, and Error-Reporting-Node may be included to provide detailed diagnostics and support automated handling or operator awareness. This mechanism ensures that errors are reported in a structured, extensible, and protocol-compliant manner without requiring separate error message types.

Table 3.5: Error DIXes.

DIX Field	Required	Use Context	Description
Result-Code	Yes	All response messages	Standardised outcome of the operation (success, error, etc.).
Extended-Result-Code	Optional	On failure	Detailed, machine-readable error condition (e.g., app-specific, vendor-defined).
Error-Message	Optional	On failure	Human-readable explanation of the error for diagnostics or UI display.
Error-Reporting-Node	Optional	On failure	Identifies the node that generated the error (e.g., "CM1", "ATC-East").
Failed-DIX	Optional	On failure	Lists specific DIX fields that were missing, invalid, or caused the failure.
Reason-DIX	Optional	Non-error events (success path)	Structured explanation for non-error-related events such as role change, session end, or failover cause.

Error information appears only in response messages (*_RESPONSE) when an error or rejection occurs. The Result-Code DIX is the primary indicator used in ATMACA to signal the outcome of a request (**Table 3.6**). It is included in all response messages to convey whether the operation was successful, partially processed, or rejected due to an error. Result codes are categorised into classes such as success, protocol errors, authorisation failures, state conflicts, and internal system issues. This classification allows agents and clients to interpret and handle responses in a consistent and interoperable manner.

Extended-Result-Code: An optional field used to provide fine-grained diagnostic information beyond what Result-Code conveys. It allows for more specific interpretation of errors, especially in application-specific or vendor-defined contexts, without overloading the standard result code space.

Error-Message: A human-readable UTF-8 string intended to explain the cause of an error in plain language. This is useful for operator dashboards, debugging, and logging, but is not meant for machine interpretation or logic branching.

Error-Reporting-Node: Identifies the node (typically by NodeName) that generated or detected the error. This allows distributed systems to trace the origin of the failure, especially when messages are relayed across multiple agents or platforms.

Failed-DIX: A grouped field containing one or more DIX elements that caused the request to fail. This helps pinpoint specific missing or invalid fields (e.g., DestConnAddr, Session-ID) and supports automated diagnostics or validation logic on the receiving side.

Table 3.6: Result Code DIXes.

Code	Name	Category	Description
1000	SUCCESS	Success	Request was processed successfully.
1001	SUCCESS_NO_OPERATION	Success	Request was valid, but no action was required.
2000	INVALID_REQUEST	Protocol Error	Malformed or semantically incorrect request.
2001	UNSUPPORTED_COMMAND	Protocol Error	The command or message type is not recognised.
2002	MISSING_MANDATORY_DIX	Protocol Error	Required DIX field(s) are missing from the message.
2003	INVALID_DIX_VALUE	Protocol Error	One or more DIX fields contain invalid or out-of-range values.
2004	FAILED_VALIDATION	Protocol Error	Request failed structural or schema validation.

3000	NOT_AUTHORIZED	Authorisation Error	Request denied due to insufficient privileges or access policy.
3001	CONTEXT_ACCESS_DENIED	Authorisation Error	Access to the specified context is not permitted.
3002	ROLE_ASSIGNMENT_DENIED	Authorisation Error	CM Agent rejected a requested role change or takeover.
4000	CONTEXT_NOT_FOUND	State/Conflict Error	The specified context could not be located or has expired.
4001	SESSION_NOT_FOUND	State/Conflict Error	The specified session is unknown or no longer valid.
4002	SESSION_ALREADY_EXISTS	State/Conflict Error	A session with the same ID already exists.
4003	CONTEXT_ALREADY_EXISTS	State/Conflict Error	Context creation failed because the context already exists.
4004	STATE_CONFLICT	State/Conflict Error	Operation cannot be carried out due to current system or context state.
5000	INTERNAL_ERROR	Internal/System Error	An internal agent or system failure occurred during processing.
5001	DOWNSTREAM_TIMEOUT	Internal/System Error	No response received from downstream node or agent.
5002	TRANSPORT_FAILURE	Internal/System Error	Network-level failure or connection loss during transaction.
5003	RETRYABLE_FAILURE	Internal/System Error	Temporary issue encountered. Retry may succeed.
9000+	VENDOR_DEFINED_<X>	Vendor/Experimental	Reserved range for vendor-specific or experimental error codes.

Reason-DIX: Unlike error-specific fields, Reason-DIX is used to explain non-error state transitions such as normal session termination, role changes, or failovers. It provides a structured context about why an event occurred, including a code, the source of the change, a description, and an optional timestamp. It supports transparency and auditability in successful but meaningful state changes.

3.1.4 ATMACA Addressing Schema Overview

The ATMACA protocol uses a structured, hierarchical addressing schema to uniquely identify and route messages between operational and supporting entities, enabling consistent communication, service discovery, and role coordination across ATM and Airline Operational Control (AOC) domains. Each address encapsulates semantic components such as functional role, deployment scope, geographic location, and operational domain, making it both human-readable and machine-resolvable. This addressing model is organised into two primary categories. Section 3.1.4.1 defines how airborne units (e.g., flights identified by call signs or flight numbers) and ground-based sectors (managed by ATC facilities) are represented within the addressing and metadata framework. The term “Operational Entity” in ATMACA refers to both flight-level constructs and sector-level constructs, enabling unified role modelling and context-aware routing. Section 3.1.4.2 introduces the naming scheme for intermediate nodes and backend servers that support protocol operations, including CM Agents, ATC Agents, Application Servers, and ATM coordination platforms. These supporting entities are named based on their functional role and deployment scope, such as facility, area, FIR, or region, and serve as core infrastructure for session management, context handling, and service delivery.

3.1.4.1 Operational Entity Addressing

ATMACA uses a structured, domain-based addressing schema to uniquely identify communication endpoints across both ATM and AOC environments (**Figure 3.4**). This schema enables consistent, hierarchical naming for ground-based ATC units, facilitating scalable routing, service discovery, and access control. Within the ATC infrastructure, addresses are organised by operational domain (i.e., Airport, Terminal Manoeuvring Area (TMA), and Enroute) and are composed of specific elements such as the ATC function (e.g., tower, ground, approach), airport or FIR identifiers (using IATA or ICAO codes), a country identifier, and a network domain suffix (atm or aoc). For instance, the address tower.lhr.gb.atm refers to a tower controller at London Heathrow in the ATM domain.

In parallel, ATMACA supports structured flight addressing to uniquely represent airborne entities from both tactical and operational perspectives. In the ATM domain, flight addresses use the aircraft call sign, followed by the operator's name, country identifier, and ATM as the network domain (e.g., thy2ab.turkish.tr.atm). In the AOC domain, addresses are instead based on the flight number, such as tk1953.turkish.tr.aoc, allowing alignment with airline dispatch and backend systems. These address formats ensure globally unique, semantically meaningful identifiers across air and ground domains, providing the foundation for reliable message routing, agent discovery, and cross-domain interoperability.

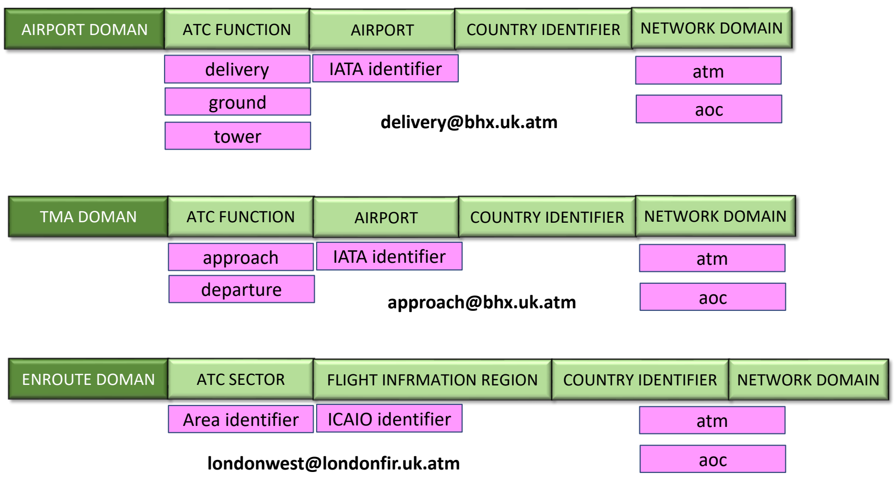


Figure 3.4: ATMACA Operational Entity Addressing Schemes.

Figure 3.4 presents the structured addressing format for operational entities in ATMACA, covering Airport, TMA, and Enroute domains, based on ATC function, location identifiers (IATA/ICAO), country codes, and network domains (ATM/AOC) (**Table 3.7**).

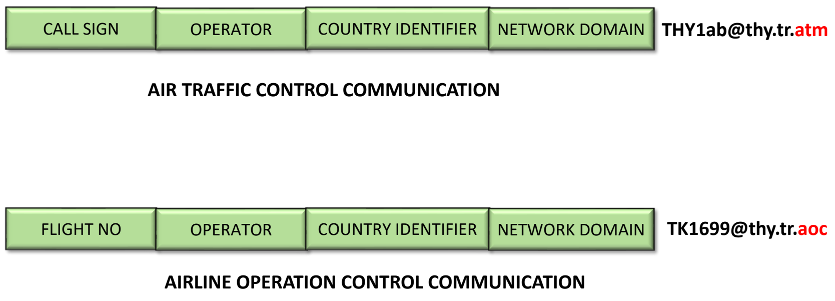


Figure 3.5: ATMACA Flight Addressing Formats for ATM and AOC Domains.

Figure 3.5 depicts the structured addressing schemes for flights in the ATMACA network, distinguishing between ATM domain addresses based on call signs and AOC domain addresses based on flight numbers, each including operator, country code, and network domain.

Table 3.7: Operational Entities.

Entity Type	Sample Address	Meaning
ATC – Tower	tower.lhr.gb.atm	Heathrow Tower Controller (UK, ATM domain)
ATC – Ground	ground.jfk.us.atm	JFK Ground Tower (US, AOC domain)
TMA – Approach	approach.ist.tr.atm	Istanbul Approach Control (Turkey, ATM)
Enroute – Sector A	sectorA.edgg.de.atm	Sector A in EDGG FIR (Germany, ATM domain)
Flight – ATM Address	thy2ab.turkish.tr.atm	Turkish Airlines aircraft with tactical call sign
Flight – AOC Address	tk1953.turkish.tr.aoc	Turkish Airlines flight number in AOC domain

3.1.4.2 Supporting Entity Addressing

In addition to operational entities shown in **Table 3.7** such as aircraft and ATC sectors, ATMACA defines structured addressing conventions for supporting entities (intermediate nodes and backend servers that enable core protocol operations). These include CM Agents, ATC Agents, Application Servers, and ATM Servers, each addressed according to its functional role and deployment scope. CM Agents are deployed at the facility level, managing context state, user presence, role transitions, and local failover mechanisms for a specific ATC unit such as a ToWeR (TWR), Area Control Center (ACC), or APProach control unit (APP). ATC Agents operate at the area level, supporting multiple facilities by managing session lifecycle, transport continuity, and service routing within that operational region. Application Servers, such as those hosting DFIS or GRO, are typically deployed per area as well, providing service-specific functionality aligned with regional ATC responsibilities. ATM Servers, which handle inter-domain coordination, policy enforcement, or cross-border data management, are generally deployed at the regional level, which may correspond to a Flight Information Region (FIR), a national airspace boundary (state), or a broader multi-state domain.

Table 3.8: ATMACA node-based addressing model.

Node Type	Deployment Scope	Address Format	Example Address	Description
ATC Agent	Per **Area** (controls multiple facilities)	atcagent.[area].[country].[domain]	atcagent.ankarea.tr.atm	ATC Agent managing session and connection logic across Ankara area
CM Agent	Per **Facility** (TWR, APP, ACC)	cmagent.[facility].[country].[domain]	cmagent.eddfapp.de.atm	CM Agent managing peer presence and role transitions at Frankfurt Approach
ATM Server	Per **FIR**, **Country**, or **Continent**	atmserver.[region].[country].[domain]	atmserver.edyy.de.atm atmserver.eu.eu.atm	Central ATM system managing airspace-wide policies and coordination
Application Server	Per **Area** (operational service node)	[service].[area].[country].[domain]	cpdlc.istarea.tr.atm	Application server (e.g., CPDLC, DFIS) supporting services within IST Area

In the ATMACA addressing model, these deployment scopes are categorised into three hierarchical levels: facility, area, and region. A facility refers to a discrete operational ATC unit and serves as the scope for CM Agent deployment (**Table 3.8**). An area represents a grouping of facilities under the control of a single ATC Agent. A region denotes a higher-order deployment context, typically used for ATM-level services that span large geographic boundaries. Each supporting entity is assigned an address using the consistent format [role].[scope].[country].[domain], ensuring semantic clarity, addressability, and protocol-wide interoperability across the ATMACA network.

3.2 Context, Session and Connection

This section describes the integrated runtime communication framework of the ATMACA protocol, which is built on three interdependent layers: the Transport (Connection) Layer, the Session Control Layer, and the Context Awareness Layer (**Figure 3.6**). Together, these layers provide robust, flexible, and scalable mechanisms for maintaining continuous communication between mobile and stationary nodes in dynamic airspace environments.

By coordinating these layers, ATMACA supports resilient, context-aware communication ensuring seamless service continuity during mobility events, airspace transitions, and varying network conditions.

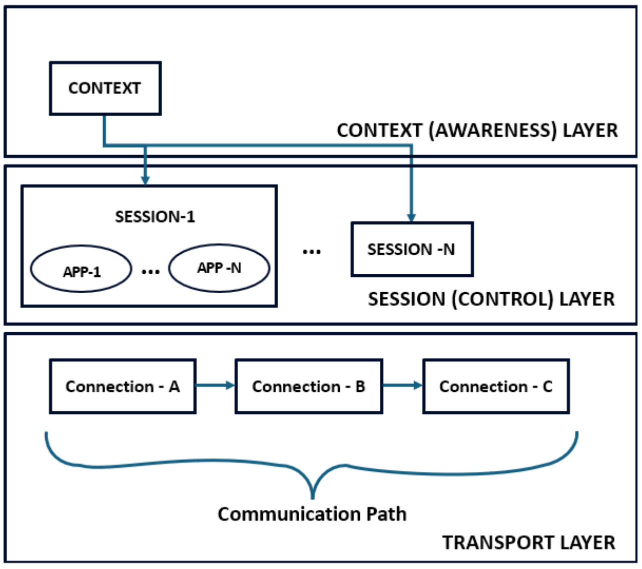


Figure 3.6: Layered Structure of ATMACA's Communication Model.

Figure 3.6 illustrates the ATMACA three-layered communication model including Transport, Session, and Context layers, each layer aligning with a specific aspect of mobility and communication lifecycle management. At the foundation lies the Transport Layer, where one or more physical or logical connections (e.g., over TCP, UDP, or SCTP) establish the communication pathways between nodes. This layer represents physical communication links. Connection management ensures service continuity during dynamic network changes supporting Terminal Mobility, and in certain configurations, also enabling aspects of Service Mobility by maintaining active communication paths as services transition between hosts or platforms.

The Session (Control) Layer aggregates the Transport Layer connections to build and maintain persistent, application-agnostic communication sessions between two endpoints. A session represents an active communication instance, and its continuity across network or agent transitions enables support for Session Mobility allowing the system to adapt in real time to changes while preserving communication integrity. Moreover, session-level control enables Service Mobility, allowing active application services to move across agents or endpoints without requiring reinitialisation, which maintains uninterrupted service.

At the top, the Context (Awareness) Layer encapsulates metadata and operational state, including user roles, controlling units, and contextual bindings. A context represents a communication entity, such as a mobile client (e.g., an aircraft) or a stationary client (e.g., an ATC workstation). Through context tracking and role management, this layer ensures User Mobility enabling service continuity as users move across operational domains or platforms.

Together, the three layers shown in **Figure 3.6** allow ATMACA to separate connection-level behaviours from higher-level application and operational concerns, delivering robust, flexible, and mobility-resilient communication in complex and dynamic air traffic environments. The ATMACA base protocol provides both connection and session core management capabilities while DLCM provides context awareness for both session and connection management in addition to context management capabilities.

3.2.1 Context Management Overview

Context Management in the ATMACA protocol is responsible for maintaining the operational state and metadata associated with users, workstations, services, and roles throughout the communication lifecycle. It enables the protocol to make intelligent, adaptive decisions about session routing, role assignment, and service continuity, particularly in dynamic environments where users or platforms may transition between facilities, control areas, or network segments.

A "context" in ATMACA encapsulates key attributes such as the current operational role (e.g., Controlling, Monitoring, or Mirroring), sector or facility associations, service configurations, and status indicators. This contextual information is continuously synchronised between agents, clients, and ATM servers, allowing seamless coordination during handovers, failovers, or multi-role operations.

Through robust context management, ATMACA ensures that communication remains aligned with operational needs supporting flexible control delegation, collaborative ATC environments, and mobility-driven service delivery across distributed systems.

3.2.2 Session Management Overview

Session Management in ATMACA provides a persistent communication framework that allows multiple applications to operate over one or more underlying network connections without disruption. Sessions serve as the logical binding between clients and services, enabling communication continuity even as transport paths change due to mobility, failovers, or network reconfiguration.

Each session encapsulates a set of applications, maintains their communication state, and is uniquely identified by a Session ID. This design allows ATMACA to decouple the session from specific connections, supporting seamless transitions between access points and transport layers. Session Management also coordinates closely with Context Management to align operational roles, user identity, and mobility status with ongoing communication.

Through this layered approach, ATMACA ensures that data exchange remains reliable, stateful, and resilient, enabling advanced ATM applications like CPDLC and DFIS to function smoothly across dynamic and distributed environments.

3.2.3 Connection Management Overview

The ATMACA Protocol uses a versatile set of communication methods designed to ensure reliable message exchange, service continuity, and efficient resource utilisation within aeronautical communication networks. These methods are designed to meet the dynamic demands of air traffic management systems, ensuring secure and uninterrupted data flow across various network environments.

ATMACA's connection management system establishes and maintains stable communication links:

- **Persistent Connections:** ATMACA maintains long-lived connections for continuous data flow between nodes. This method reduces handshake overhead and supports ongoing session management.
- **Heartbeat Mechanism:** Periodic heartbeat messages are exchanged between nodes to verify active connections and promptly detect network failures.
- **Failover Support:** By using SCTP's multi-homing capabilities, ATMACA dynamically switches to alternate communication paths during link disruptions, ensuring uninterrupted service.

3.2.3.1 Transport Layer Support

ATMACA supports multiple transport protocols to ensure flexible and robust communication:

- **TCP:** Ensures reliable, connection-oriented communication with error recovery mechanisms, making it suitable for critical data exchanges such as CPDLC and DLCM services.
- **UDP:** Provides lightweight, connectionless communication for time-sensitive data that tolerates occasional packet loss, ideal for broadcasting flight information updates.
- **SCTP:** Enhances reliability with multi-homing and multi-streaming capabilities, ensuring uninterrupted connectivity during link failures and improving data flow efficiency.

3.2.3.2 Port Assignments

The ATMACA base protocol operates over dedicated ports to distinguish between various aeronautical services:

- **DLCM:** Port 5910 (TCP/UDP)
- **CPDLC:** Port 5911 (TCP/UDP)
- **DFIS:** Port 5912 (TCP/UDP)

Port 5910 is the primary ATMACA base protocol port for initiating communication before message exchanges occur. All ATMACA nodes, including clients, agents, and servers, must be capable of listening on port 5910 for both TCP and UDP to ensure compatibility with initial connections. An ATMACA node may initiate connections from a source port other than 5910 but must always be prepared to receive incoming connections on port 5910 for both TCP and UDP.

3.2.3.3 Connection Establishment and Recovery

If no active transport connection exists with a peer, ATMACA nodes attempt to establish a dynamic peer connection. There is a reconnection behaviour that is controlled by the Tc timer (time to reconnet), which is recommended to be set at 30 seconds to balance responsiveness and network efficiency. However, exceptions apply in cases where a peer has explicitly terminated the connection and indicated that it does not wish to communicate, in which case no further reconnection attempts should be made. When attempting to reconnect with a peer, ATMACA nodes should prioritise TCP as the preferred transport protocol when no specific transport method is indicated.

3.3 DLCM

The DLCM function is the foundational application layer of the ATMACA protocol, serving as the runtime coordination environment that bridges aeronautical applications such as CPDLC, DFIS, and GRO, with the underlying communication, session, and context management infrastructure. It enables context-aware, role-driven, and mobility-resilient operation of services across both stationary and mobile nodes. Evolving from the limited DLIC in the ATN/OSI model, DLCM in ATN/IPS introduces a broader, service-oriented architecture capable of sustaining seamless operation under dynamic network and operational conditions.

DLCM delivers an integrated set of services critical to distributed air traffic communication. These services include:

- **Connection Management:** Handles the establishment and maintenance of transport-layer connections, manages transitions between Air Traffic Services Units (ATSUs), and enables seamless handovers across network domains.
- **Session Management:** Supports the dynamic creation, maintenance, and termination of application-level sessions, ensuring communication continuity throughout the operational lifecycle.
- **Context Management:** Maintains logical associations between clients and their operational metadata, including IP addresses, ATSU roles, session bindings, and user identity. It also supports context updates, transfers, and disassociations as required during operational transitions.
- **Mobility Management:** Enables transparent mobility across FIR boundaries through automated reassignment of agents, ATSUs, and transport paths. It supports user, session, terminal, and service mobility as part of the ATMACA mobility framework.
- **Presence Management:** Tracks and disseminates the online/offline status and role-specific availability of all participants, including ATC units and aircraft, to facilitate real-time awareness and coordinated communication among operational entities.
- **Service Delivery Management:** Orchestrates the provisioning, invocation, and lifecycle control of aeronautical services such as DFIS, and GRO, based on predefined policies, service priorities, and operational triggers, ensuring reliable and context-aware service continuity across network domains.

DLCM is composed of three key software modules: the Context, Session, and Connection Managers, each responsible for handling specific dimensions of the protocol lifecycle management. The Context Manager handles user mobility through context association and role tracking; the Session Manager ensures session mobility via dynamic establishment, maintenance, and transfer of application sessions; and the Connection Manager manages terminal and service mobility, including session continuity across ATSUs, although service mobility may also be supported by the Session Manager depending on configuration. These modules may be deployed individually or in various combinations: when the Session Manager and Connection Manager are co-located, the platform is referred to as an ATC Agent; a platform hosting only the Context Manager is called a CM Agent; and when all three modules are integrated into a single platform, it is referred to as a CM/ATC Agent. This modular and flexible design supports both centralised and distributed deployments, enhancing the scalability and fault tolerance of the protocol.

The next section outlines the architectural role of DLCM, its key service functions including connection and session lifecycle management, context synchronisation, application binding, and seamless mobility handling, as well as its interaction with client applications. It also describes how DLCM manages runtime events such as handovers, rebinding, and failovers, forming the operational backbone of distributed, digital air traffic management services under ATMACA.

3.3.1 Role and Position in the Architecture

The DLCM layer plays a central role in the ATMACA protocol architecture as the foundational application responsible for managing application-to-network interactions in a modular, role-aware, and context-sensitive manner. It operates as a runtime middleware environment, sitting between client-facing aeronautical applications and the underlying communication infrastructure including connection, session, and context layers.

DLCM does not host or implement the business logic of individual services like CPDLC or DFIS. Instead, it exposes a standardised set of functions that these applications rely on to establish, maintain, and recover their network presence and operational state. It manages essential elements such as application binding, session multiplexing, role-based service access, mobility handling, and failover recovery, ensuring that applications remain stable and reachable as operational and network conditions change.

Architecturally, DLCM interacts with:

- Client applications via a standardised service interface that abstracts low-level session and transport logic.
- Agents and ATM servers to coordinate session routing, role tracking, and context handovers.
- Transport and session layers to monitor network state, manage reconnections, and maintain service continuity.

By mediating between these components, DLCM ensures that communication remains robust, synchronised, and aligned with the operational role and context of each user or node. It enables highly dynamic, distributed ATM environments, such as FIR-crossing flight operations, multi-role controller setups, and mobile ATC units, to function with consistency and resilience.

This architectural positioning allows DLCM to serve as the operational core of the ATMACA runtime, enabling seamless integration of future applications and services while maintaining backwards compatibility with legacy data link models.

3.3.2 Core Service Capabilities

The DLCM application in the ATMACA protocol is responsible for managing the establishment, maintenance, and termination of data link communication sessions, encompassing the DLIC as a core component. DLCM ensures the secure exchange of identity, context, and connectivity information necessary for initiating and sustaining CPDLC and other data link services.

Its primary scope includes the following key aspects:

- **Initial Contact and Identification:** DLCM facilitates the first point of contact between the aircraft and the ATSU by managing the DLIC process, which includes exchanging key identifiers such as the aircraft’s Flight ID and ATSU information. This ensures mutual awareness prior to session establishment.

- **Context Association Management**: DLCM is responsible for linking terminal context including IP address, ATSU assignment, and session metadata with the appropriate ATSU or agent. This association is critical for ensuring accurate routing, session targeting, and coordination across distributed nodes.
- **Session Preparation and Control:** DLCM prepares the communication environment by aligning key parameters such as supported services, session timeouts, and encryption settings. It then dynamically manages session activation, continuity, and termination as aircraft move between ATSUs or network paths.
- **Peer-to-Peer Relationship Establishment:** DLCM establishes routing relationships between aircraft, ATSUs, and agents by exchanging control information and propagating routing metadata. This enables efficient handover, agent coordination, and distributed communication state awareness.
- **Presence and Status Monitoring:** DLCM continuously monitors the status of associated clients and terminals. If a disconnection, mobility event, or timeout occurs, DLCM updates or clears relevant records and triggers service continuity mechanisms as needed.
- **Service Delivery**: DLCM ensures the reliable invocation, routing, and lifecycle management of application-layer services such as CPDLC, DLIC, DFIS, and GRO. It dynamically provisions these services based on session context, operational role, and current connectivity status. Service requests are evaluated against the current mobility state, agent availability, and supported capability sets to determine optimal delivery paths. DLCM handles retransmissions, timeout recovery, and quality monitoring, while also enforcing service-specific policies and sequencing constraints. This guarantees that critical aeronautical services are delivered in a timely, consistent, and context-aware manner across diverse network environments and operational transitions.
- **Mobility Support:** DLCM supports terminal mobility by managing IP reassignment, context handover, and automatic reassociation of ATSUs and agents. It ensures that applications maintain persistent sessions and roles even as clients move across FIRs or switch access technologies (e.g., VHF ↔ SATCOM).

These integrated capabilities form the runtime backbone of the protocol, enabling client applications to operate reliably across evolving operational conditions. The following subsections provide a detailed breakdown of each DLCM service domain. Mobility management is the fourth core capability of the DLCM service layer within the ATMACA architecture. It ensures continuity of service and communication as nodes or users change their point of attachment or role within the network. This capability supports dynamic, distributed air traffic environments by managing transitions across physical, logical, and service contexts while maintaining session and data integrity throughout.

The ATMACA Mobility Management framework is designed to support four distinct types of mobility:

- **User Mobility:** Managed by the Context Management module. It enables users (e.g., controllers or pilots) to dynamically associate themselves with different contexts without losing operational state or role.
- **Session Mobility:** Handled by the Session Management module. It allows communication sessions to persist and transfer between different controllers or agents, ensuring uninterrupted service during handovers.
- **Terminal (Device) Mobility:** Supported by the Connection Management module. It tracks and maintains connectivity as clients (e.g., flight deck or ATC terminals) physically move between network zones or ATC Agents.

- **Service Mobility:** Enabled by the infrastructure routing and registry systems. It ensures that services can follow the user or in some cases, originate from mobile nodes (e.g., aircraft-based servers) by dynamically resolving and rerouting service endpoints as mobility events occur.

By modularising these mobility aspects across the stack, ATMACA provides resilient, scalable mobility support for a wide range of deployment scenarios. It ensures that critical services, context roles, and communication sessions remain intact and synchronised, even as users and nodes traverse different regions, facilities, or control domains.

All four mobility types, user, session, terminal, and service, ultimately converge on a single architectural goal: ensuring service continuity. Regardless of whether a controller changes context, an aircraft shifts between ATC sectors, a terminal moves across network zones, or a service endpoint relocates, the ATMACA system is designed to preserve the operational flow. By coordinating context ownership, session routing, connectivity, and service availability, the mobility management framework guarantees that ongoing communication and functionality are never interrupted. This principle of uninterrupted service underpins the resilience and scalability of ATMACA's mobility-aware architecture.

ATMACA’s mobility management relies on a set of well-defined protocol messages that ensure seamless continuity as users, sessions, terminals, and services move across the network. For most types like user mobility, session mobility, and service mobility, the system builds existing messages already defined under context, session, and connection management. These messages are simply reused to support mobility without adding protocol overhead. However, terminal mobility requires more active monitoring of network conditions, so it introduces a few new message types designed specifically for detecting disconnections and updating transport details in real time.

4 SOFTWARE ARCHITECTURE

The ATMACA protocol is implemented through a modular and layered software architecture designed for high portability, runtime flexibility, and clean separation of concerns. Each module in the system encapsulates a specific functional responsibility, ranging from low-level transport and connection handling to protocol parsing, session management, context coordination, and application integration. These components interact through clearly defined interfaces, event-based messaging, and shared service contracts, enabling seamless integration across diverse deployment scenarios such as CM Agents, ATC Agents, and application servers.

This section outlines the architectural composition of the ATMACA software stack, describing the primary modules, their runtime interactions, class responsibilities, and deployment models. It also introduces the internal layering principles, extensibility mechanisms, and concurrency models that allow the system to remain robust and adaptable under operational constraints such as mobility, failover, and dynamic reconfiguration.

The ATMACA software architecture is designed to support modular, distributed protocol processing across a variety of deployment scenarios, including embedded systems, ground infrastructure, and cloud-based application platforms. Its core logic is divided into well-defined modules, each responsible for a specific layer of protocol processing: from transport and session management to context awareness, application integration, and mobility coordination.

The modular structure promotes separation of concerns, parallel development, and easier testing and deployment. Components interact through clearly defined interfaces and data flows, allowing flexibility in deployment whether components are co-located (e.g., in a CM/ATC Agent) or distributed across separate agents.

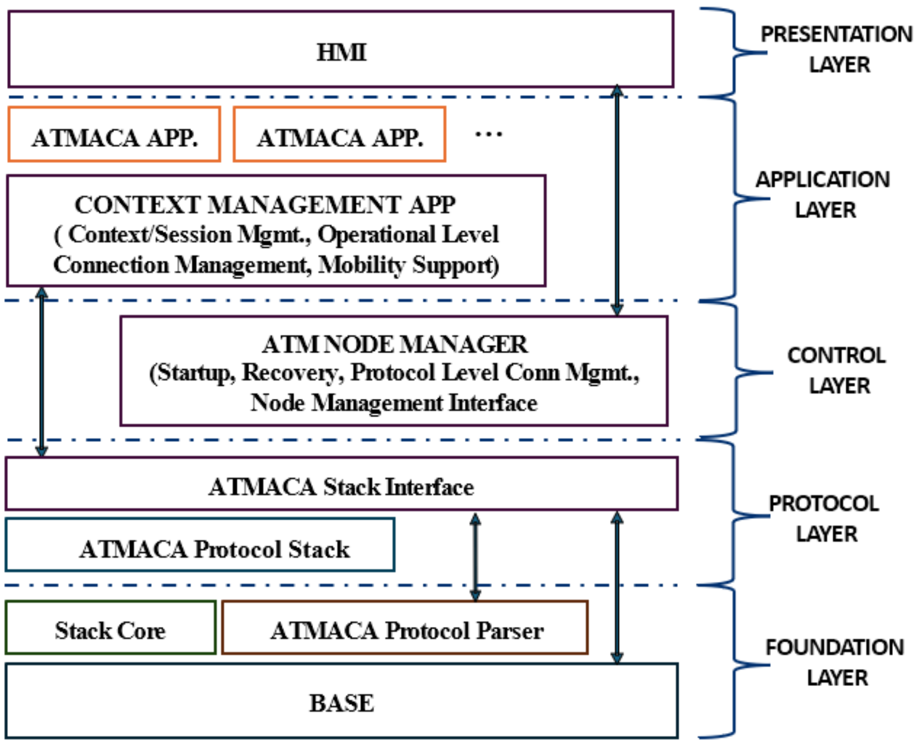


Figure 4.1: ATMACA Software Architecture Layer Model.

Figure 4.1 presents the layered software architecture of ATMACA, illustrating how the system is structured from the foundational base and protocol stack up to control, application, and presentation layers, enabling modular communication, session management, and user interaction.

The ATMACA software architecture is designed with a layered modular structure, ensuring clear separation of concerns, platform portability, and protocol extensibility. At the foundation lies the Base layer, which provides essential OS abstraction components such as threads, mutexes, and socket interfaces. This layer encapsulates platform-specific differences, allowing the upper layers to remain hardware- and OS-agnostic. Built atop this foundation, the Stack Core implements generic mechanisms for network connectivity, message input/output, and connection management.

The ATMACA Protocol Stack layer then defines protocol-specific behaviours and state machines, implementing ATMACA's messaging logic, session control rules, and operational constraints on top of the stack core. Complementing this, the ATMACA Protocol Parser, along with its builder counterpart, offers a robust set of utilities for encoding, decoding, and validating ATMACA messages, forming the heart of protocol message processing. These parser tools are used directly by both the stack and the application layer. Sitting on top of the protocol interface is the ATM Node Manager, which orchestrates the system lifecycle. It is responsible for initialising all application-level components, handling node startup and recovery, and exposing management interfaces for operational visibility. The Node Manager initialises the Context Management Application layer, which resides above the Node Manager, orchestrating high-level functions such as session lifecycle management, peer coordination, connection supervision, and mobility handling.

ATMACA Applications operate at the top of the stack, interfacing with the parser and management components to send, receive, and interpret protocol messages in alignment with user services such as CPDLC and DFIS.

At the base lies the Foundation Layer, which comprises the BASE, Stack Core, and ATMACA Protocol Parser components. This layer provides essential platform services, including memory management, logging, timers, and protocol parsing, forming the runtime bedrock of the system.

4.1 ATMACA Base Layer

ATMACA Base is a modular, extensible C++ infrastructure framework designed to support the development of high-performance, network-centric systems. It provides a rich set of components for multithreading, network communication, inter-process messaging, dynamic module loading, protocol handling, configuration management, metrics, and logging. ATMACA is highly suitable for applications in telecommunications (like Authentication, Authorisation, and Accounting (AAA) servers), real-time services, embedded systems, and scalable backend infrastructure where performance, control, and reliability are critical.

The architecture of ATMACA emphasises cross-platform compatibility (Linux and Windows), efficient resource management, and fine-grained system control. It abstracts away the low-level complexities of system calls, socket operations, and thread management while retaining flexibility and transparency. At its core, ATMACA Base is designed to be thread-safe, protocol-agnostic, and adaptable to various communication patterns including TCP, TLS, and UDP.

4.1.1 Threading and Concurrency

At the heart of ATMACA's concurrency model is the Thread class, which provides a unified interface for POSIX and Windows threading. Threads are further coordinated using synchronisation primitives such as Mutex, RecursiveMutex, and ReadWriteLock. To simplify safe lock handling, classes like ScopedLock and ScopedLock<T> ensure automatic acquisition and release of locks, enhancing exception safety and code clarity.

Advanced threading components include the Scheduler, which manages and audits threads, tracks job durations, and identifies potential deadlocks. The TimerService class enables scheduling periodic or one-shot timers with callback execution, crucial for event-driven systems and protocol timeouts (**Figure 4.2**).

4.1.2 Socket and Network Abstraction

ATMACA Base features a powerful and extensible socket abstraction layer. The base Socket class supports both stream-oriented (TCP, TLS) and datagram (UDP) communication. Derived classes such as TCPSocket, TCPTLSSocket, and UDPSocket encapsulate protocol-specific behaviour, including non-blocking I/O, connection management, and multicast support (**Figure 4.2**).

The framework introduces a socket factory pattern via SocketMaker and its derivatives (TCPSocketMaker, TCPTLSSocketMaker, and UDPSocketMaker). This allows sockets to be instantiated dynamically based on protocol strings and configurations. The UDPListener class demonstrates asynchronous event reception, executing in a separate thread and invoking application callbacks.

4.1.3 Reference Counting and Memory Safety

Instead of relying on standard smart pointers, ATMACA implements an intrusive reference counting system through RefCounted, Ref, and RefTemplate<T> (**Figure 4.2**). This allows for fine-grained control over object ownership, memory safety, and lifetime management, particularly important in long-lived systems or shared communication resources. The pointertemplate complements this approach by acting as an auto-cleaning container for raw pointers, preventing memory leaks in dynamic collections.

4.1.4 Dynamic Module and Protocol Message Handling

ATMACA provides support for runtime dynamic modules via the DynModule class. This allows the framework to load shared libraries at runtime, facilitating plugin architectures or dynamic protocol extensions. Alongside this, the ProtocolMessage and RawData classes serve as generic containers for protocol data, enabling encapsulation and transport of binary payloads across different modules (**Figure 4.2**).

4.1.5 Configuration Management

The framework includes a simple, yet effective configuration parser named SimpleConfigHandler. This class reads attribute=value formatted configuration files, supporting repeated keys, typed lookups (bool, int, double), and fallbacks. This enables application modules to be parameterised without recompilation and supports flexible deployment environments.

4.1.6 Logging, Auditing, and Metrics

Observability in ATMACA is powered by the Logger, Audit, and Metric components. These modules enable fine-grained logging of system events, periodic audits for health checks or diagnostics, and quantitative performance tracking (**Figure 4.2**). Logs can be filtered, redirected, and integrated with higher-level monitoring systems. Audit supports callback registration to facilitate periodic internal checks of system state or user-defined conditions.

4.1.7 Utilities and Helpers

Several utility modules enrich the framework. The Utility class contains system-level helpers such as error handling, IPC pipes, and sleep functions. ObjectQueue offers a high-throughput, thread-safe queue used for event passing or producer-consumer models. AAAUri implements a URI parser compliant with RFC 3588 (**Figure 4.2**).

ATMACA also includes a custom lightweight XML parser (XmlParser), which is DOM-based and supports programmatic traversal, attribute management, and node mutation. This can be used to parse or emit protocol messages, configurations, or command instructions formatted as XML.

4.1.8 Class Interactions and System Design

ATMACA Base stack is designed to be layered and loosely coupled. Threads, sockets, configuration, and scheduling operate independently but integrate through shared interfaces and utilities (**Figure 4.2**). Dynamic socket factories simplify protocol extensions without changes to core logic. Timers and schedulers allow decoupling of application logic from timing constraints. Reference counting and scoped locks enhance memory and concurrency safety.

Each module is built to be testable, mockable, and replaceable, fostering unit testing and modular development. The architecture allows protocol stacks and communication servers to be composed of reusable components with minimal duplication or cross-dependency.

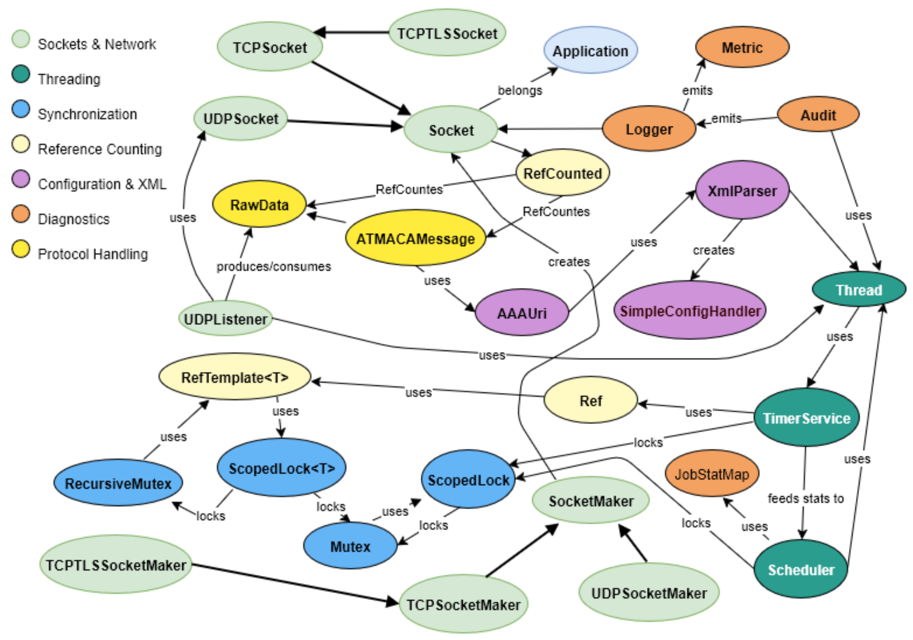


Figure 4.2: Dependency Graph of ATMACA Protocol Software Components.

4.1.9 External Integration & Module Interactions

ATMACA Base provides the low-level services that other systems or applications build upon. It offers reusable, encapsulated functionality for threading, networking, logging, and configuration (**Figure 4.2**).

4.1.9.1 Outbound Interactions (ATMACA Base using other modules)

- Dynamically load protocol modules or plugins via DynModule. These can be developed externally and loaded at runtime.
- Call user-defined callbacks (e.g., TimerService, Audit, UDPListener) for events, timeouts, or incoming packets.
- Parse external configuration files using SimpleConfigHandler to adapt its behaviour without code changes.
- Log to external systems using Logger which is easily extendable to integrate with syslog, Elasticsearch, Logstash, Kibana, or Prometheus exporters.
- Send/Receive data over the network, interacting with clients, servers, or other services using standard protocols (TCP, UDP, TLS).

4.1.9.2 Inbound Interactions (Other modules using ATMACA Base)

- Use ATMACA Base as a transport and infrastructure library, importing headers and linking to its binaries.
- Create and manage socket connections using SocketMaker
- (e.g., Socket* s = SocketMaker::CreateSocket("tcp", ...)).
- Register custom logic into timers (TimerService), event loops (UDPListener), or audit tasks (Audit).
- Use the XML parser (XmlParser) for lightweight XML config or messaging formats.
- Extend the framework by adding new socket types, protocol handlers, or metrics collectors.

4.2 ATMACA Protocol Parser

The ATMACA Parser is the core mechanism within the ATMACA Base Protocol responsible for parsing, validating, and constructing protocol messages from raw data streams. It transforms low-level network bytes into semantically structured AtmacaMessage objects, enabling higher-level components to work with typed, validated information (**Figure 4.3**). Drawing inspiration from Diameter protocol concepts, ATMACA uses a flexible DIX (Data Interchange eXchange) system to define and encode message fields, which are parsed based on a dictionary-driven approach. The parser design emphasises extensibility, allowing new DIX types and message formats to be integrated without impacting existing logic.

The ATMACA Parser is a robust and extensible engine for decoding ATMACA Base Protocol messages. Its layered architecture, combining header parsing, dictionary-based DIX interpretation, factory-driven object creation, and grammar validation, provides both power and flexibility. Whether handling standard messages, nested grouped DIXs, or unknown fields, the parser maintains reliability and extensibility. This design ensures that the system can adapt to future protocol evolution, application-specific requirements, and complex message formats, all while preserving a clean and maintainable codebase.

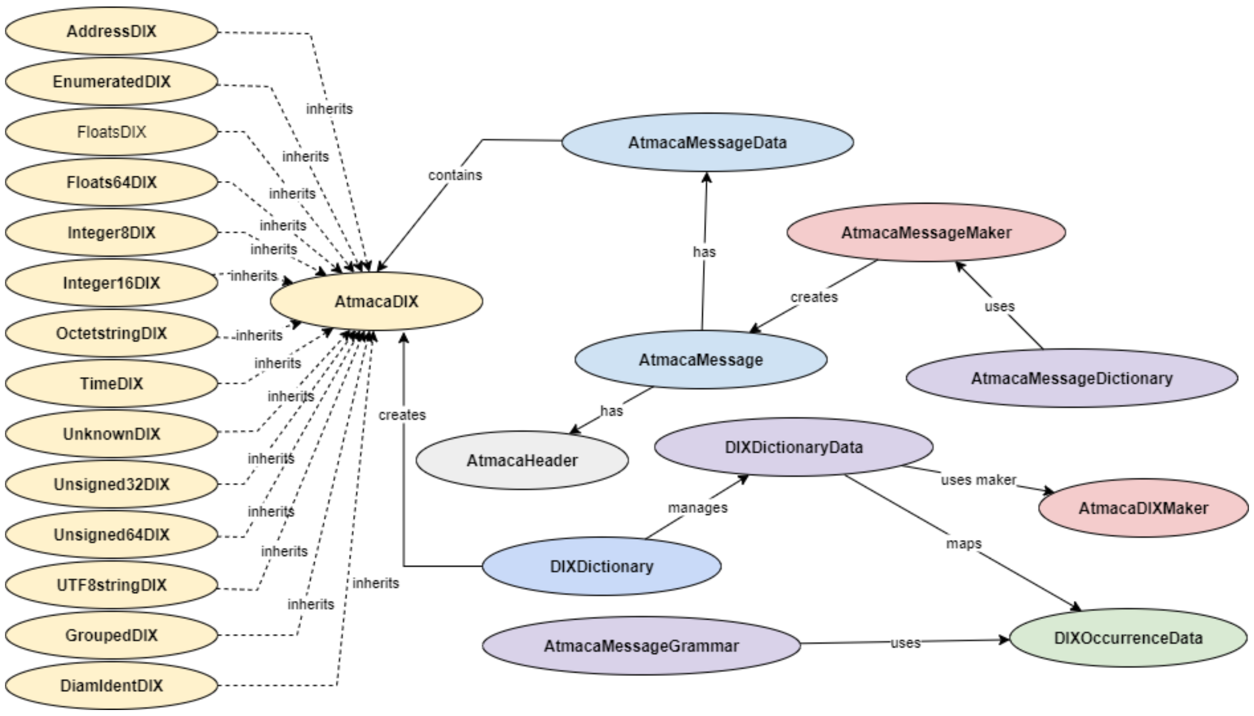


Figure 4.3: Class and Data Structure Relationships in ATMACA Message Handling.

Figure 4.3 illustrates the object-oriented relationships among ATMACA message components, showing how various DIX types inherit from a base class, and how message creation, dictionary usage, and header associations are managed within the ATMACA messaging framework.

4.2.1 Message Structure and Parsing Process

Parsing begins with the AtmacaMessage class, the primary structure used to represent protocol messages. This class encapsulates not only the raw bytes, but also the header fields, parsed DIX elements, and any associated metadata. The method ParseRawDataForMessage initiates the parsing process by interpreting the fixed-length header. This includes extracting version, application ID, command code, request ID, message length, and several flags that determine message type and routing behavior.

Once the header is parsed, the parser queries the AtmacaMessageDictionary to identify the correct handler for the message. This dictionary matches messages based on a tuple of (commandCode, applicationId, isRequest) and returns an AtmacaMessageMaker responsible for creating the appropriate AtmacaMessage subclass. This ensures that each message can apply specialised logic while still conforming to the base structure.

Following handler creation, the parser invokes ParseDixPart to step through the DIX-encoded payload. Each DIX segment includes code, flags, length, optional vendor ID, and value. These are interpreted in sequence, with each segment mapped to a handler class using the DIXDictionary （**Figure 4.3**).

4.2.2 DIX Dictionary and Makers

The DIXDictionary is the registry that binds each (dixCode, vendorId) pair to a DIXDictionaryData object. Each entry defines the DIX's data type, associated flags, human-readable name, and a reference to an AtmacaDIXMaker, a factory that constructs instances of the appropriate DIX handler class (**Figure 4.3**).

This design allows the parser to remain agnostic to the underlying DIX types, using the maker system to dynamically instantiate handlers at runtime.

AtmacaDIXMaker is the base interface for all DIX factories. Integer32DIXMaker, Float64DIXMaker, and GroupedDIXMaker subclasses implement a CreateDIXHandler method, which produces a new instance of a specific DIX type based on its code, vendor, flags, and dictionary metadata. These maker instances are usually implemented as singletons, ensuring lightweight reuse across the parser.

4.2.3 DIX Types and Data Interpretation

Each DIX is represented by a subclass of AtmacaDIX, such as Integer32DIX, UTF8StringDIX, GroupedDIX, or TimeDIX. These classes encapsulate type-specific parsing and data representation logic. For example, Integer32DIX knows how to read 4-byte integers, while GroupedDIX contains lists of nested DIXs and handles recursive parsing (**Table 4.1**).

Grouped DIXs are especially powerful as they can contain other DIXs, forming a nested hierarchy of information. They implement additional methods to retrieve inner DIXs by code or name, manage multiple occurrences, and enforce group-specific grammar rules. Each DIX subclass implements parsing methods such as SetDataFromRaw, CopyRawData, and PrintData to handle serialisation and debugging.

All known data formats are supported, including:

- Basic types like integers (Integer32DIX, Integer64DIX), unsigned values, floats (Float32DIX, Float64DIX)
- Strings via OctetStringDIX, UTF8StringDIX, and DiamIdentDIX
- Complex types such as TimeDIX (NTP timestamp) and GroupedDIX
- Fallback types using UnknownDIX, which assumes octet-string formatting for unknown codes

Table 4.1: DIX Types, Their Corresponding Classes, and Maker Classes.

DIX Type	Class	Maker Class
Integer (32/64)	Integer32DIX, Integer64DIX	Integer32DIXMaker, Integer64DIXMaker
Unsigned (32/64)	Unsigned32DIX, Unsigned64DIX	Unsigned32DIXMaker, Unsigned64DIXMaker
Float	Float32DIX, Float64DIX	Float32DIXMaker, Float64DIXMaker
String	OctetStringDIX, UTF8StringDIX, DiamIdentDIX	Corresponding DIXMakers
Time	TimeDIX	TimeDIXMaker
Grouped	GroupedDIX	GroupedDIXMaker
Enumerated	EnumeratedDIX	EnumeratedDIXMaker
Unknown	UnknownDIX	UnknownDIXMaker

4.2.4 Future Extensions

The ATMACA Parser is designed with extensibility at its core, allowing developers to integrate new DIX types with minimal friction. To add a new DIX type, the developer simply needs to create a subclass of the AtmacaDIX base class, implementing the specific parsing, serialisation, and data handling logic for the new format. Alongside this, a corresponding AtmacaDIXMaker must be implemented to act as a factory for the new DIX type. Once the maker is created, it must be registered in the global DIXDictionary, associating it with a unique DIX code and vendor identifier. This modular approach ensures that new features or protocol extensions can be introduced without modifying the core parser logic.

The system also includes built-in resilience mechanisms. When the parser encounters a DIX that is not defined in the dictionary, it falls back to using the UnknownDIX class. This generic handler assumes an octet string data type and allows the parser to safely handle and retain unknown elements without failing or discarding data. This capability is critical for forward compatibility, allowing the protocol to evolve while maintaining support for legacy infrastructure.

Another important feature is the parser’s dual-mode support. All components of the parser are capable of operating in both parsing and message-building contexts. By setting the forParsing flag to true, the system enters parsing mode to decode raw binary streams into structured message objects. Conversely, when forParsing is false, the same components can be used to construct binary messages from structured data, facilitating use cases such as message generation, testing, and simulation. This symmetrical design promotes reuse, consistency, and simplicity across the protocol stack.

4.2.5 Grammar and Validation

Beyond basic decoding, the parser performs structural validation using AtmacaMessageGrammar, a class that defines the expected layout of DIXs for each message type. For each DIX, the grammar defines constraints like minimum and maximum occurrences, whether it is mandatory, and if it must appear in a fixed position. These rules are enforced after parsing, ensuring the message complies with its schema.

Grammar entries are held in DIXOccurrenceData objects and mapped via (dixCode, vendorId) keys. This grammar enforcement ensures both interoperability and protocol correctness, especially important in security-sensitive or standards-compliant deployments. GroupedDIX elements also validate their inner DIXs recursively using grammar data stored in the DIXDictionaryData.

4.2.6 Parsing Flexibility and Fallback

One of the parser's strengths is its graceful handling of unknown or future DIX elements. If a DIX code is not found in the dictionary, the parser creates a default UnknownDIX object using the UnknownDIXMaker. This allows partially understood messages to be processed without triggering errors or dropping data, enabling forward compatibility and vendor-specific extensions.

Furthermore, the parser supports both parsing mode (forParsing = true) and message building mode (forParsing = false), using the same structure for serialisation and deserialisation. This symmetry simplifies testing, debugging, and round-trip message handling.

4.2.7 Extending the Parser

Adding support for a new DIX type or message structure is straightforward. Developers can subclass AtmacaDIX to implement parsing for a new data format, create a matching AtmacaDIXMaker, and register it in the DIXDictionary. Likewise, custom messages can be supported by subclassing AtmacaMessage, implementing a corresponding AtmacaMessageMaker, and registering it in the AtmacaMessageDictionary.

This plug-in style architecture ensures that protocol extensions or application-specific types can be added without modifying the core parsing logic. It also allows independent teams to develop and maintain their own dictionary sets, improving modularity and maintainability.

4.3 ATMACA Core Stack

The ATMACA Protocol Stack serves as the backbone of communication within the ATMACA system, managing peer connections, session handling, transaction routing, and transport-layer integration. Designed for extensibility and reliability, the stack operates with both connection-oriented and connectionless transport protocols including TCP, SCTP, TLS, and UDP. Its modular design comprises core components such as ScPeer, ScPeerManager, ScSession, and TransactionManager, all interacting under a robust configuration environment provided by the StackCore. By abstracting protocol-specific behaviours and enforcing clean separation of concerns through callback mechanisms and reference-counted objects, the stack supports scalable signalling management across dynamically discovered and statically configured peers. The system is also equipped with timeout mechanisms, Finite-State Machine (FSM) handling, TLS context management, and logging facilities to ensure high availability and observability in production environments.

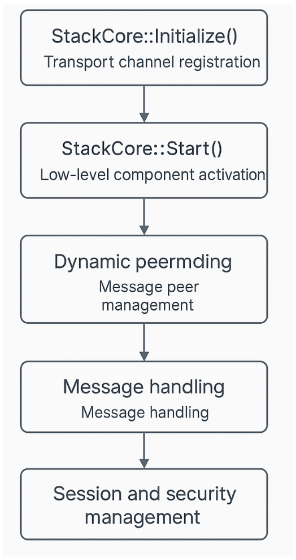


Figure 4.4: ATMACA StackCore Initialisation and Runtime Workflow.

Figure 4.4 shows the initialisation and operational flow of the ATMACA StackCore, from transport channel registration and component activation to dynamic peer management, message processing, and session/security management.

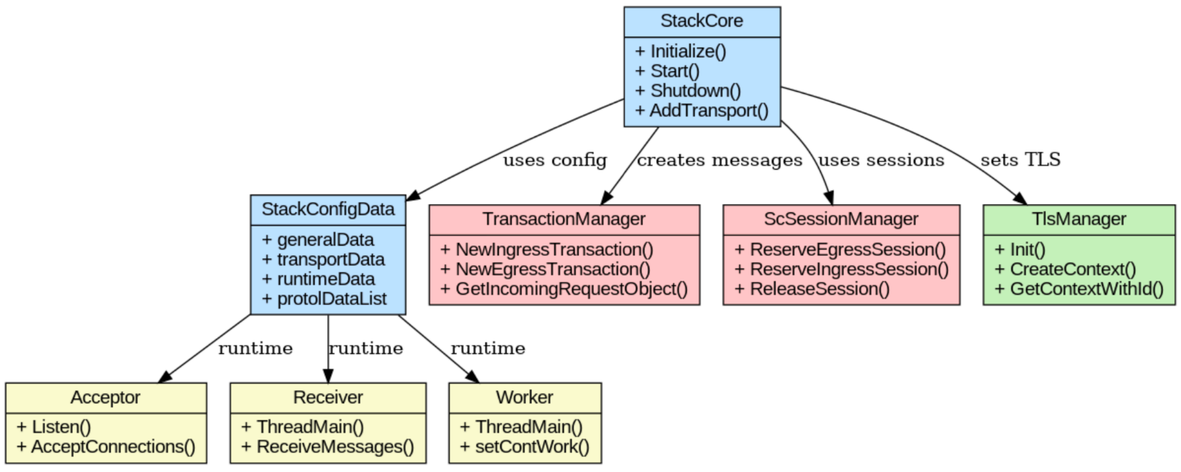


Figure 4.5: ATMACA StackCore Component Interaction Diagram.

Figure 4.5 shows the internal structure and runtime interactions of the ATMACA StackCore, detailing how it initialises configuration, manages transactions, sessions, and TLS contexts, and coordinates runtime roles such as acceptors, receivers, and workers.

4.3.1 Stack Initialisation

The lifecycle of the ATMACA Protocol Stack begins with its initialisation phase, orchestrated primarily by the StackCore class. Upon startup, StackCore::Initialize() sets internal flags, configures logging, and prepares the stack to register transport channels. The stack supports multiple transports means (TCP, UDP, and TLS) configured through StackConfigData::AddTransport, which instantiates TransportComponent objects with associated IP addresses, ports, and protocols. Each transport layer is maintained within StackTransportData, allowing for flexible lookup and management throughout runtime.

4.3.2 Stack Startup

Once transports are added, StackCore::Start() is called. This function activates low-level components including the Acceptor (to handle incoming TCP connections), an array of Receiver threads (for dispatching data from sockets), and a pool of Worker threads (to process application-level logic). These elements are initialised using the configuration stored in StackRuntimeData, and their creation is dynamically adjusted based on configuration parameters such as receiverCount and workerCount defined in StackGeneralData.

4.3.3 Peer Management and FSM Control

The stack dynamic behaviour is driven by the concept of peers, each represented by the ScPeer class. A peer encapsulates a remote node with which the local stack communicates. Peers can be created statically using configuration files or dynamically when an incoming connection is accepted. Peer creation is handled via ScPeerMakerand registered in ScPeerManager, which centralises the control of all peer connections. The ScPeerManager uses an FSM defined in ScPeerFSM to manage connection states such as INIT, WAIT_CONN_ACK, and OPEN, and it ensures that reconnections and retransmissions are handled gracefully.

EUROPEAN PARTNERSHIP

Co-funded by
the European Union

4.3.4 Message Handling and Transactions

Message handling is centered around the ScMessage class. When a message arrives, it is received by a Receiver thread, which parses the message and determines whether it is a request or a response. Requests are handled by creating a new transaction object via the TransactionManager, which tracks all active message exchanges using internal maps for both ingress and egress directions. Egress messages, sent as requests to other peers, are linked with callbacks to handle asynchronous responses using ScResponseCallbackFunctionTable. Ingress transactions are created when incoming requests are received and stored with a ScIncomingRequestHandle, which the application uses to send back a response.

4.3.5 Session Management

Sessions are managed independently through the ScSessionManager, which tracks long-lived communication channels across multiple transactions. Sessions are uniquely identified using a combination of the host ID, a timestamp, and a local counter, in accordance with practices inspired by RFC 3588. The session manager provides methods to reserve, retrieve, and release sessions while distinguishing between ingress and egress directions. Each session object is represented by an instance of ScSession, which maintains its own state and links to the associated peer.

4.3.6 TLS and Secure Transport Support

For security, the stack supports TLS through the TlsManager component. It maintains SSL contexts, certificates, and transport-level encryption policies for both incoming and outgoing connections. TLS contexts are created per transport using identifiers, and Socket objects are initialised accordingly to enable secure communications.

4.3.7 Observability and Extensibility

Throughout runtime, all stack operations are observable and configurable through runtime flags, logging hooks, and callbacks. Developers can register a ScLogObserver to redirect logs into custom sinks such as application UIs or monitoring tools. Additionally, developers can extend core behaviours by subclassing major interfaces like Transaction, ScPeer, or ProtocolData to insert custom logic, metrics, or event tracking mechanisms.

4.3.8 Developer Notes and Extension Points

The ATMACA Protocol Stack is designed for extensibility, modularity, and clean separation between protocol logic and application-level behaviour. It offers multiple integration points where developers can extend the stack functionality, monitor runtime activity, and customise message and session handling based on protocol-specific requirements.

At the peer level, developers can subclass the ScPeer class to embed custom peer behaviours such as application-specific connection handling, peer validation, or heartbeat logic. Each peer operates under a finite-state machine defined in ScPeerFSM, and transitions can be observed or influenced by overriding state callbacks or injecting interceptors into message flow logic.

For message processing, the ScMessage class provides a generic message abstraction that can be extended or wrapped to embed higher-layer semantics. Developers building a new protocol or signaling logic on top of ATMACA can use ScMessageFactory to centralise the creation of custom message objects, formatters, or dispatchers. Message callbacks are provided through ScResponseCallbackFunctionTable, which allows asynchronous response handling with user-defined logic.

Transaction control is abstracted via the Transaction and TransactionManager classes. The system maintains a clear distinction between ingress and egress transactions and tracks them through reference-counted handles. If a protocol requires complex transaction matching or custom timeout behavior, developers can extend the Transaction class or override lifecycle management through hooks available in the transaction manager. The design enables non-blocking, parallel message flows without the need for manual correlation logic.

Session management is exposed through ScSessionManager and ScSession, which support long-lived state across multiple transactions. This is particularly useful for applications that require session-based state tracking (e.g., Diameter-like protocols). Developers can extend session objects to hold custom data, enforce idle timeouts, or implement session cleanup policies.

Secure communication is handled through the TlsManager, which provides fine-grained control over SSL/TLS contexts, including per-transport certificates and verification settings. If needed, developers can integrate mutual TLS, update cipher policies, or plug in custom certificate validation logic by wrapping the SSL layer during context creation.

For logging and observability, the stack uses a pluggable logging framework that allows applications to register a ScLogObserver. This lets developers route logs to custom dashboards, files, or telemetry systems. Additionally, the stack exposes runtime state information through internal flags in StackRuntimeData and metrics like active transaction counts, which can be used to monitor system health and throughput.

To integrate custom protocols, the ProtocolData base class is available as an extension point. Implementations can register themselves via StackConfigData::AddProtocolData, enabling centralised access to protocol-specific configurations and objects.

Overall, the ATMACA stack is built to support clean extension through subclassing, dynamic registration, and modular configuration, making it ideal for adapting to new signalling protocols, enterprise middleware needs, or research-grade protocol experimentation.

4.4 ATMACA Protocol Stack

The ATMACA Protocol Stack is a modular and extensible communication framework built to support high-performance, connection-oriented or datagram-based messaging between distributed systems. Inspired by concepts from Diameter, MQTT, and RADIUS, it facilitates message transmission, peer discovery, state machine orchestration, and robust transaction handling. At the heart of this stack are peer entities (AtmacaPeer), which manage connections and handle the transmission of protocol messages.

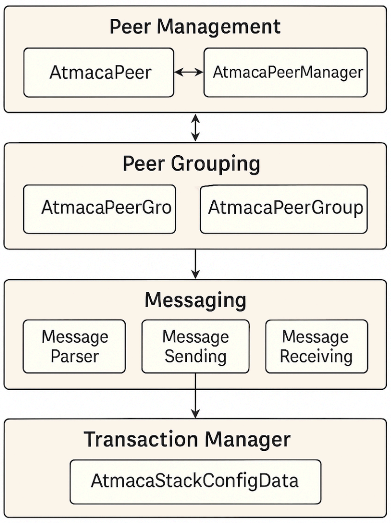


Figure 4.6: ATMACA Messaging and Peer Management Architecture.

Figure 4.6 illustrates the layered architecture of ATMACA's messaging system, showing how peer management, peer grouping, message parsing and exchange, and transaction configuration are organised for reliable communication.

4.4.1 Peer Management & Lifecycle

The AtmacaPeer class extends the base ScPeer and acts as the communication anchor for the ATMACA protocol. Each peer handles lifecycle events including connection establishment, request/response message processing, and disconnection. Peers use state machines to ensure robust connection handling, including transitions such as STATE_WAIT_CONN_ACK, STATE_I_OPEN, and STATE_R_OPEN. A built-in state machine timer (smTimerId) helps detect issues like handshake timeouts and triggers reconnection logic as needed.

Each peer instance holds an AtmacaPeerConfig object that carries parameters such as origin host/realm, shared secrets, and socket settings. Dynamic configuration is supported, allowing for on-the-fly creation of peer identities especially useful when accepting unknown connections over TCP or handling datagrams over UDP.

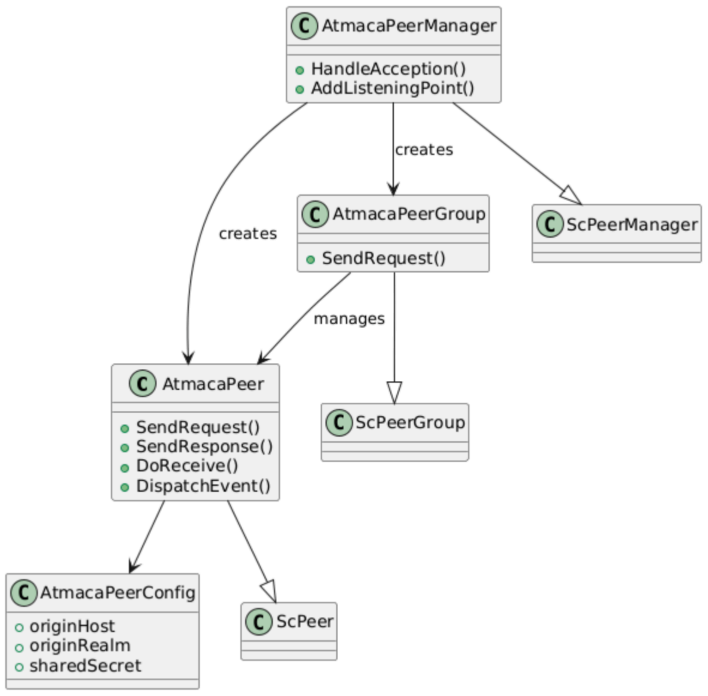


Figure 4.7: ATMACA Peer Management Class Diagram.

Figure 4.7 depicts the class relationships within ATMACA's peer management framework, showing how peer managers create and configure peers and peer groups, manage requests and responses, and maintain secure peer configurations.

4.4.2 Peer Manager & Grouping

All peers are managed centrally through the AtmacaPeerManager, which orchestrates peer creation, socket listening, message acceptance, and dynamic peer generation. When a remote client connects, the manager either assigns an existing AtmacaPeer handler or creates one dynamically, particularly in the case of UDP or non-preconfigured clients.

The AtmacaPeerGroup class facilitates peer grouping, enabling the system to send requests to any available connected peer. If no peers are connected, messages are safely queued in an "orphan transaction" pool until a valid transport is available.

4.4.3 Messaging & Transaction Control

Message transmission in ATMACA is request-response oriented. Each outgoing request is assigned a unique request ID using ObtainRequestId() and encapsulated into a ScMessage object. The peer sends the message directly if the connection is live or queues it for delivery once connected. Upon reception of a message, DoReceive() handles reading from either stream (TCP) or datagram (UDP) sockets and hands off the raw data to the dispatcher.

The dispatcher (DispatchEvent) classifies messages and calls either HandleReceivedRequest() or HandleReceivedResponse(). The request path creates a transaction using TransactionManager, invokes a callback, and allows the application to respond using SendResponse(). The response path matches incoming replies to active transactions and invokes associated response handlers.

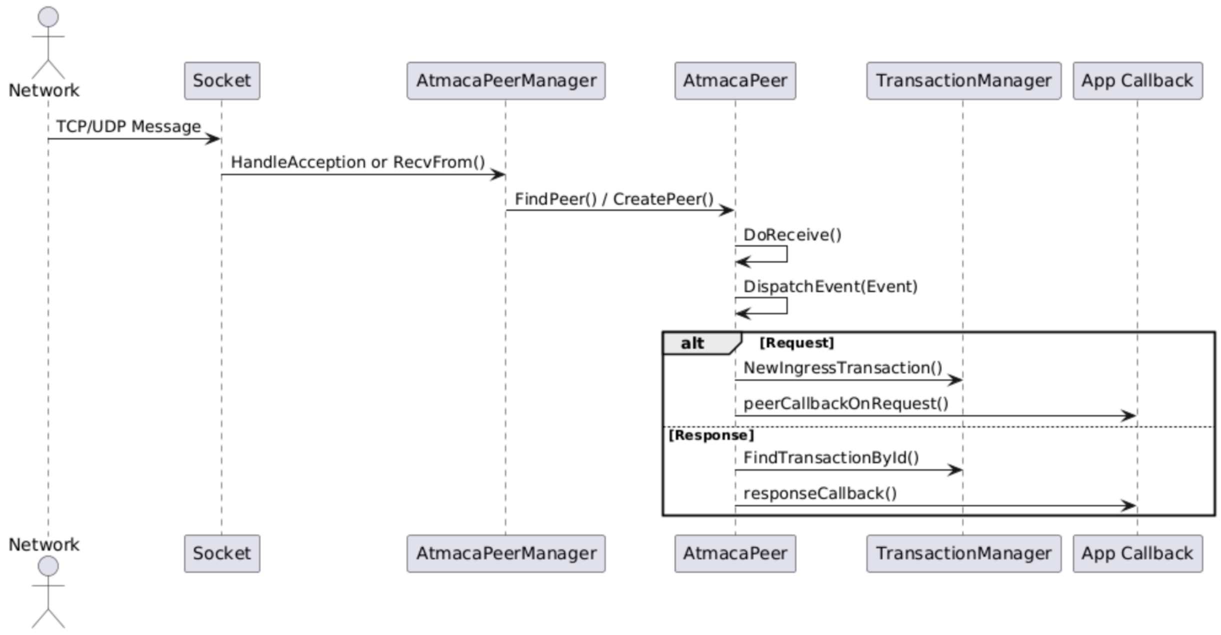


Figure 4.8: Message Dispatch Flow.

Figure 4.8 shows the sequence of operations in ATMACA when handling a TCP/UDP message, including peer identification or creation, message reception, event dispatching, and request-response transaction handling across the peer and application layers.

4.4.4 Developer Notes

Developers integrating with ATMACA can extend the system by subclassing components such as AtmacaPeer, AtmacaMessage, or custom application-level handlers. Every peer operates asynchronously, and the system is designed to accommodate TLS extensions, although secure transports are currently stubbed out in logs for future implementation. The framework's modular design ensures flexibility for various message protocols or system roles.

4.4.5 Configuration & Metrics

Configuration is centralised in the AtmacaStackConfigData, which holds general data (e.g., vendor ID, product name), transport parameters (e.g., origin hosts, host IPs), and runtime metrics. These metrics are initialised per peer and can be used to track connection attempts, malformed messages, or system counters like unrecognised peer attempts.

The stack uses macros like ATMACA_CONFIG_GENERAL() and ATMACA_CONFIG_TRANSPORT() to provide global access to configuration data throughout the system. This simplifies integration with other components and ensures a clean separation between core logic and configurable parameters.

4.5 ATMACA Protocol Stack Interface

The ATMACA Protocol Stack Interface forms a vital extension of the core ATMACA Protocol Stack, introducing layers and mechanisms tailored for advanced routing, peer management, message observation, and application-specific behaviour. This layer enhances flexibility, observability, and control of peer-to-peer interactions, especially in dynamic air traffic management environments where resilience and adaptability are crucial.

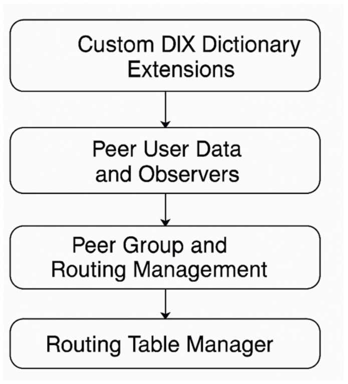

Custom DIX Dictionary Extensions
Peer User Data and Observers
Peer Group and Routing Management
Routing Table Manager

Figure 4.9: ATMACA Routing Management Hierarchy.

Figure 4.9 outlines the hierarchical flow of routing management in ATMACA, beginning with custom DIX dictionary extensions and progressing through peer data handling, group routing control, and final route table management.

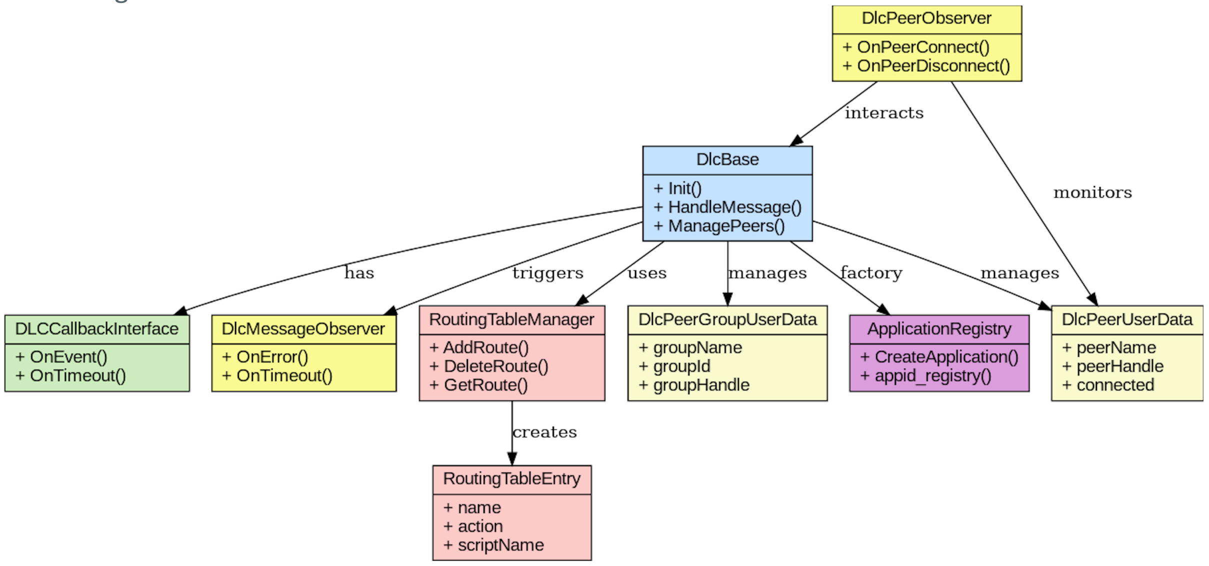


Figure 4.10: Class Relationship Diagram for DLC Routing and Peer Management.

Figure 4.10 depicts the interactions and responsibilities among core classes in ATMACA's Data Link Communication (DLC) module, showing how the base class manages peers, triggers routing actions, monitors peer states, and interacts with observers, registries, and user/group data handlers.

4.5.1 Custom DIX Dictionary Extensions

One of the key responsibilities of the ATMACA Protocol Stack is enriching the Diameter-inspired messaging system with additional AVP definitions. This is achieved through two main classes: DlcBaseDIXDictionary and DlcDIXDictionary. The DlcBaseDIXDictionary sets up dictionary entries for various operational contexts, node attributes (e.g., origin/host types), flight data (e.g., callsign, aircraft registration), facilities (e.g., FIRs, domains), ATM-specific message components, sectors, CPDLC elements, and adjacent agent metadata. Each AVP is registered using specific DIX types such as UTF8StringDIXMaker and Unsigned32DIXMaker. Meanwhile, DlcDIXDictionary defines more general-purpose AVPs following the Diameter model, such as Session-Id, Origin-Host, Destination-Host, grouped types like Proxy-Info, and vendor-specific fields. These definitions extend the core dictionary with application-specific semantics and routing logic.

4.5.2 Peer User Data and Observers

Peers in the ATMACA Protocol Stack are tracked with detailed metadata using the DlcPeerUserData class. This structure holds the peer's name, connection status, whether it was dynamically created, its type (static or dynamic), associated application ID, transport type, node role/type, and detailed state machines (e.g., created, connected, disconnected). This setup enables rich introspection and lifecycle management for peer entities.

The system uses dedicated observer classes like DlcPeerObserver and DlcMessageObserver to hook into the peer and message events. These classes provide static methods to bind callback functions into the stack, such as connect, disconnect, timeout, failover, and message error handling. Combined with the DlcCallbackInterface, applications can implement logic for real-time logging, dynamic peer creation responses, message retries, and failover actions.

4.5.3 Peer Group and Routing Management

The class DlcPeerGroupUserData extends the peer logic into groups, enabling the handling of multiple peers as a single logical entity. This is particularly useful for load balancing, redundancy, and coordinated failover. The class tracks the group name, unique ID, and a vector of associated peer handles.

Message routing within the ATMACA Protocol Stack is handled by the RoutingTableManager, which provides multilevel hierarchical routing logic. Incoming messages are evaluated based on the sender's peer name, command code, application ID, and destination realm. Matching rules (routes) are encapsulated as RoutingTableEntry instances and stored in layered maps. This architecture allows for precise control over message dispatching, supports dynamic reconfiguration, and even uses filters and scripts for programmable routing behaviour.

The routing logic can respond differently depending on configuration (e.g., default vs. custom routes) and is protected using reader-writer locks for concurrency. Routes can be added, updated, deleted, or queried through the API or CLI interfaces.

4.5.4 Overall Role and Flexibility

The ATMACA Protocol Stack significantly boosts the adaptability of the system, allowing tailored AVPs, robust message routing, and fine-grained callback logic to respond to complex networking and communication scenarios. It serves as a middleware that links application-layer services to core messaging functions and provides hooks for custom business logic, security enforcement, and performance monitoring.

4.6 Control Layer

The Control Layer is represented by the ATM Node Manager, which acts as the central orchestrator of the system’s lifecycle. Positioned directly above the protocol interface, it is responsible for managing the initialisation and supervision of all application-level components.

The Node Manager Module serves as the foundational component within the ATMACA software stack, responsible for managing the core runtime state and operational configuration of the local node. It provides a unified interface for initialising, configuring, and controlling the node across its functional roles whether operating as a Server, Agent, or Client. The module encapsulates node-wide behaviour through the abstract base class AtmNode, from which specialised classes inherit depending on the deployment scenario.

The Node Manager handles structured datasets representing key operational entities such as facilities, sectors, flights, and peer nodes. These entities are dynamically managed at runtime and serve as the core reference for context coordination, session routing, and domain awareness. In server deployments, the Node Manager maintains shared tables (e.g., sectorTable, flightList, facilityTable) and peer registration data. In agent roles, it tracks hosted facilities, adjacent areas, and embedded service components like CM or ATC Agents. In client deployments, the module distinguishes between Mobile Clients (supporting data authority handovers and session mobility) and Fixed Clients (serving in tower or station roles with local scope).

4.6.1 Internal Collaborations

The Node Manager module does not operate in isolation, it relies on a set of integrated components and shared data services to manage the full operational context of an ATMACA node. These internal collaborations represent structured interactions with key subsystems responsible for configuration management, transport connectivity, software metadata, and domain-specific operational data such as peer relationships, facilities, sectors, and flights. Together, these elements form the execution environment that allows the Node Manager to track node state, enforce protocol logic, and support runtime coordination across agent and client roles. This subsection outlines the main internal collaborators that underpin the Node Manager's functionality.

The AtmNode class serves as the central orchestrator for these operations and collaborates with a number of key support components to maintain and operate the node runtime environment. These include:

- **nodeBaseConfiguration**: Accessed via baseConfData, this component manages configuration file paths, directory structures, and startup parameters.
- **atmSoftwareInfo**: Stores runtime metadata such as software version, label, and debug flags used in diagnostics and message headers.
- **DlcNetwork**: Provides the transport layer interface for sending and receiving ATMACA messages, including watchdog supervision and connection lifecycle handling.
- **PeerData**: Represents the connected peer nodes and their associated state, used in both context and session decision-making.
- **Domain-specific datasets**: Including SectorData, FacilityData, and FlightData, which form the operational core of node-scoped logic and role resolution.

The module supports both static configuration and dynamic reconfiguration, offering control points for startup, shutdown, recovery, and state introspection. It acts as the primary service container in the protocol runtime and is the first layer initialised in any ATMACA-based node deployment. Through inheritance, encapsulated role-specific behaviour, and well-defined interfaces, the Node Manager provides the structural backbone that enables all other modules including context and session agents to operate in a cohesive, state-aware manner.

4.6.2 Node Startup and Configuration

ATMACA nodes are initialised using structured configuration files that define their identity, operational parameters, and data provisioning environment. These files are delivered in a standardised JavaScript Object Notation (JSON) format and are loaded during startup by the Node Manager module. Each configuration captures the complete runtime profile of a node, enabling it to establish communication, join the ATMACA network, and load domain-specific resources such as facility tables, peer lists, and protocol message catalogues. This initialisation process ensures consistent behaviour across deployments while allowing flexibility to tailor configurations based on the node's assigned role (e.g., ATM Server, ATC Agent, CM Agent, or Client). The structure is modular, separating core identity from dynamic settings and external data dependencies, making it easier to manage, track versions, and extend in future protocol updates.

Each ATMACA node begins its lifecycle by loading a structured initialisation file in JSON format, which defines its runtime identity, operational parameters, and data provisioning paths. These files provide a standardised and extensible mechanism for bootstrapping the node, eliminating the need for manual configuration or hardcoded settings. The structure of the file is consistent across all node types whether a server, agent, or client, but specific fields and file references may vary based on the node's functional role. The initialisation file is logically divided into three primary sections:

- ATM-NODE-DEFINITION: Captures the node's static identity and network bindings, including type, role, name, and communication endpoints.
- ATM-NODE-CONFIGURATION: Defines runtime behaviours such as communication port settings, transport types, timeouts, peer connection thresholds, and fault handling policies.
- ATM-NODE-PROVISION: Points external resources needed by the node at startup, including facility tables, message lists, logger files, and common datasets.

This layered structure promotes a clean separation between identity, behaviour, and provisioning, making it easier to manage and version control node configurations. The use of JSON enables easy integration with orchestration tools and allows centralised provisioning of large-scale deployments, ensuring consistency across all ATMACA-enabled components.

4.6.3 Air Traffic Management (ATM) Server

The ATM Server node represents a region- or FIR-level backend component responsible for managing airspace-wide coordination services, inter-agent discovery, policy propagation, and domain-level resolution. Its initialisation file includes all general node definition elements but also references server-specific datasets used for routing, airspace mapping, and CPDLC integration.

A typical ATM Server configuration includes:
- A node role set to ATM_SERVER
- Realm-wide addressing (e.g., global.atm) and local address bindings
- Messaging and transport configuration for context and CPDLC service support
- Provisioning of airspace datasets including Facility Tables, Sector Tables, and Area Tables
- Reference files for known ATM peer nodes (global server list) and CPDLC message definitions

These files allow the ATM Server to act as a regional coordination point across multiple agents and clients, serving as a trusted relay or policy distribution node within the ATMACA architecture.

4.6.3.1 ATM-NODE-DEFINITION

Listing 4.1 defines the identity and addressing information of the ATM Server within the ATMACA network. It includes the node's unique name, type (SERVER), role (ATM_SERVER), realm, and local host bindings. This information allows peer agents and clients to discover and communicate with the server using standardised addressing.

```
{
"NodeId": 100,
"NodeName": "EUROPE_REGION",
"NodeHost": "EasternZone1@global.atm",
"NodeType": "SERVER",
"NodeRole": "ATM_SERVER",
"NodeRealm": "global.atm",
"NodeLocalAddress": "127.0.0.1",
"NodeLocalPort": "DCL_DEFAULT_PORT"
}
```

Listing 4.1: Example ATC Agent Node Configuration.

4.6.3.2 ATM-NODE-CONFIGURATION

Listing 4.2 outlines the operational parameters for transport, communication behaviour, and fault handling. It includes port assignments for protocol-specific services (e.g., DLCM and CPDLC), timeout values, retry limits, and keepalive thresholds. These parameters ensure consistent and resilient communication handling during runtime.

```
{
 "NodeDlcmCommPort": 5910,
 "NodeDlcmTransportType": 2,
 "NodeCpdlcCommPort": 5811,
 "NodeCpdlcTransportType": 2,
 "NodeMsgTimeoutValue": 10000,
 "NodeMsgTimeoutCounter": 15,
 "NodeNumberofPeers": 100,
 "NodePeerConnAttemptCounter": 10,
 "NodePeerKeepAliveCounter": 10,
 "NodeNumberofFaultRecords": 100
}
```

Listing 4.2: Example ATC Agent Node Provision.

4.6.3.3 ATM-NODE-PROVISION

Listing 4.3 specifies the file system paths and provisioning datasets used by the ATM Server during initialisation. It includes references to the facility, sector, and area tables, as well as CPDLC message catalogs and global server lists. These datasets are used to populate runtime memory, support provisioning of downstream agents, and coordinate airspace-wide services.

```
{
 "NodeConfigurationFilePath": "../../data/NodeConfigurationFiles/",
 "NodeConfigurationFileName": "AtmServerConfigurationFileEUROPE.json",
 "NodeRecoveryFilePath": "../../data/RecoveryFiles/AtmServer/",
 "NodeRecoveryFileName": "NodeRecoveryFileEUROPE.json",
 "NodeLoggerFilePath": "../../data/LoggingFiles/AtmServer/",
 "NodeLoggerFileName": "NodeLoggerFileEUROPE",
 "NodeDataFilePath": "../../data/DataFiles/AtmServer/",
 "NodeCommonFilePath": "../../data/CommonFiles/",
 "NodeFacilityTable": "AtmFacilityTable.json",
 "NodeSectorTable": "AtmSectorTable.json",
 "NodeAreaTable": "AtmAreaTable.json",
 "NodeAtmGlobalServerListFile": "AtmGlobalServerList.json",
 "NodeAtmCpdlcMessageListFile": "AtmCpdlcMessageList.json"
 "NodeApplicationList": ["DLCM"]
}
```

Listing 4.3: Example ATC Agent Node ATM Server Configuration.

4.6.4 Dataset Provisioning Workflow

In ATMACA deployments, authoritative airspace datasets, such as facility tables, sector definitions, area maps, and message catalogues, are initially managed and delivered by a centralised Management Station. These datasets are formatted in standardised JSON and are downloaded into the ATM Server during its initialisation phase. The ATM Server acts as the central repository and distributor of these datasets across the ATMACA network.

Once provisioned, the ATM Server uses these files to support downstream node provisioning during the registration and startup process of CM Agents, ATC Agents, and Client nodes. When a new node comes online and initiates its registration, the ATM Server supplies the relevant configuration subsets needed for that node to operate within the designated airspace and role. This ensures consistency in operational scope, eliminates configuration mismatches, and enables centralised versioning and updates of airspace definitions. This model allows the ATMACA network to scale in a synchronised and policy-aligned manner while reducing manual configuration efforts and runtime inconsistencies.

4.6.4.1 Area Table Description

The ATM-AREA-TABLE defines the spatial and operational scope of each ATC Agent within the ATMACA network (Error! Reference source not found.).

```
{
 "ATM-AREA-TABLE": [
   {
     "AtcAgentID": 2001,
     "AreaName": "ISTAREA",
     "AgentDatalinkAddress": "127.0.0.1",
     "AdjacentAreaList": [
       {
        "AdjacentAreaName": "ANKAREA",
        "AdjacentAgentDatalinkAddress": "127.0.0.1"
       }
     ]
   },
   {
     "AtcAgentID": 2002,
     "AreaName": "ANKAREA",
     "AgentDatalinkAddress": "127.0.0.1",
     "AdjacentAreaList": [
       {
        "AdjacentAreaName": "ISTAREA",
        "AdjacentAgentDatalinkAddress": "127.0.0.1"
       }
     ]
   },
   {
     "AtcAgentID": 2003,
     "AreaName": "EASTAREA",
     "AgentDatalinkAddress": "127.0.0.1",
     "AdjacentAreaList": [
       {
        "AdjacentAreaName": "EASTAREA",
        "AdjacentAgentDatalinkAddress": "127.0.0.1"
       }
     ]
   }
 ]
}
```

Listing 4.4: Visual Representation of Area Table in JSON format.

Each entry in the table represents an individual control area such as an FIR, regional sector, or defined ATC jurisdiction and includes the following attributes:

- **AtcAgentID:** A unique identifier assigned to the ATC Agent responsible for the area.
- **AreaName:** A human-readable label for the managed area.
- **AgentDatalinkAddress:** The data link address (typically IP or logical endpoint) that other nodes use to communicate with the agent.
- **AdjacentAreaList:** A list of neighbouring control areas and their respective agent addresses, used to support session mobility, routing decisions, and boundary handovers.

This table enables the ATM Server and ATC Agents to coordinate adjacency awareness, identify neighbouring control regions, and facilitate dynamic agent reassignment or inter-area communication. It is used by ATC Agents during startup to determine their regional scope and to initialise routing paths for managing mobile clients as they transition across boundaries. The adjacency information also supports mobility-related logic such as context rebinding, peer discovery, and handover requests.

4.6.4.2 Facility Table Description

The ATM-FACILITY-TABLE defines the operational characteristics and affiliations of each ATC facility within the ATMACA environment (Error! Reference source not found.).

```
{
 "ATM-FACILITY-TABLE": [
   {
    "FacilityID": 1000,
    "FacilityName": "SAWAIRPORT",
    "FacilityType": "CTR",
    "FacilityFIR": "ANKARA_FIR",
    "FacilityDomain": "AIRPORT_DOMAIN",
    "FacilityAreaName": "ISTAREA",
    "FacilityHost": "sawairport@saw.tr.atm",
    "FacilityRealm": "saw.tr.atm",
    "FacilityInitialContactSector": "SAWDELIVERY"
   },
   {
    "FacilityID": 1001,
    "FacilityName": "ESBAIRPORT",
    "FacilityType": "CTR",
    "FacilityFIR": "ANKARA_FIR",
    "FacilityDomain": "AIRPORT_DOMAIN",
    "FacilityAreaName": "ANKAREA",
    "FacilityHost": "esbairport@saw.tr.atm",
    "FacilityRealm": "esb.tr.atm",
    "FacilityInitialContactSector": "ESBDELIVERY"
   },
   {
    "FacilityID": 1002,
    "FacilityName": "ISTAIRPORT",
    "FacilityType": "CTR",
    "FacilityFIR": "ISTANBUL_FIR",
    "FacilityDomain": "AIRPORT_DOMAIN",
    "FacilityAreaName": "ISTAREA",
    "FacilityAreaID": 6001,
    "FacilityHost": "istairport@saw.tr.atm",
    "FacilityRealm": "ist.tr.atm",
    "FacilityInitialContactSector": "ISTDELIVERY"
   }
 ]
}
```

Listing 4.5: Visual Representation of Facility Table in JSON format.

This dataset is used by ATC Agents and the ATM Server to identify and manage facilities during session initialisation, handover processing, and message routing. Each facility entry includes attributes such as a unique identifier, facility name, associated control area, FIR, domain type (e.g., AIRPORT_DOMAIN), and initial contact sector. It also contains addressing metadata such as the facility host address and realm. Facilities are grouped under operational areas, allowing agents to efficiently manage regional responsibilities while maintaining global routing awareness.

This dataset enables CM Agents to manage context roles, peer bindings, and mobility state relative to each facility, while ATC Agents use it for session assignment, routing decisions, and inter-facility handover management. By maintaining consistent facility definitions across agents, the table ensures synchronised interpretation of operational regions and enhances coordination during client movement or multi-agent communication. The table is provisioned by the ATM Server during registration, ensuring that all nodes operate with a common and authoritative view of the ATC facility landscape.

4.6.4.3 Sector Table Description

The ATM-SECTOR-TABLE defines the structure, identity, and adjacency of air traffic control sectors within a facility, enabling spatial and procedural awareness for both agents and clients in the ATMACA environment (**Listing 4.6**). Each sector corresponds to a specific air traffic function, such as DELIVERY, GROUND, or TOWER, and includes metadata related to its facility, FIR, domain, VHF frequency, and datalink addressing. The dataset allows agents to coordinate sector-level handovers, track operational boundaries, and support dynamic session and context transitions across sector lines.

Each entry specifies sector-level identifiers (SectorID, SectorName), communication details (SectorHost, SectorDatalinkAddress, SectorVHFAddress), and adjacency mappings via the AdjacentSectorList, which defines logical neighbour relationships for seamless routing and coordination. The InitialContactSector flag indicates which sector should serve as the first point of contact for mobile clients entering a facility's jurisdiction.

The ATM-SECTOR-TABLE is provisioned by the ATM Server and downloaded to both ATC Agents and CM Agents during the node registration process.

- ATC Agents use it to manage session mobility, route messages to appropriate service endpoints, and handle inter-sector communication.
- CM Agents use sector-level information to construct and maintain context instances tied to operational sectors, and to support user mobility by tracking client presence, role continuity, and eligibility for role reassignment as users move across sector boundaries.

```
{
"ATM-SECTOR-TABLE": [
{
"SectorID": 1000,
"SectorName": "SAWDELIVERY",
"SectorType": "DELIVERY",
"SectorFacility": "SAWAIRPORT",
"SectorDatalinkAddress": "sawdelivery@saw.tr.atm",
"SectorVHFAddress": "120.5",
"SectorDomain": "AIRPORT_DOMAIN",
"SectorAreaName": "ISTAREA",
"SectorFIR": "ANKARA_FIR",
"InitialContactSector": true,
"AdjacentSectorList": [
{ "AdjacentSectorName": "SAWGROUND" },
{ "AdjacentSectorName": "ESBTOWER" }
]
},
{
"SectorID": 1001,
"SectorName": "SAWGROUND",
"SectorType": "GROUND",
"SectorFacility": "SAWAIRPORT",
"SectorDatalinkAddress": "sawground@saw.tr.atm",
"SectorVHFAddress": "121.9",
"SectorDomain": "AIRPORT_DOMAIN",
"SectorAreaName": "ISTAREA",
"SectorFIR": "ANKARA_FIR",
"InitialContactSector": false,
"AdjacentSectorList": [
{ "AdjacentSectorName": "SAWDELIVERY" }
]
},
{
"SectorID": 1002,
"SectorName": "ESBTOWER",
"SectorType": "TOWER",
"SectorFacility": "ESBAIRPORT",
"SectorDatalinkAddress": "esbtower@esb.tr.atm",
"SectorVHFAddress": "121.1",
"SectorDomain": "AIRPORT_DOMAIN",
"SectorAreaName": "ANKAREA",
"SectorFIR": "ANKARA_FIR",
"InitialContactSector": false,
"AdjacentSectorList": []
}]}
```

Listing 4.6: Visual Representation of Sector Table in JSON format.

This shared dataset ensures synchronised spatial understanding across the ATMACA network, enabling robust, role-aware, and mobility-resilient air traffic communication services.

4.6.5 Air Traffic Communication Agent

The ATC Agent is a core software entity responsible for handling session management, message routing, transport continuity, and mobility coordination across multiple facilities within a designated control area. To initialise and operate in the ATMACA network, each ATC Agent receives a configuration file in structured JSON format, which is transferred by the Management Station at deployment time. This file defines the agent's identity, operational parameters, peer communication behaviours, and the location of supporting data resources such as facility lists, global server catalogues, and protocol message definitions.

Unlike runtime provisioning datasets such as sector and facility tables, downloaded from the ATM Server during the registration process, the ATC Agent configuration file is part of the pre-provisioned static setup delivered to the node via management tooling or orchestration systems. This distinction ensures that the ATC Agent is correctly scoped, networked, and aligned with system-wide policies before initiating dynamic interactions within the ATMACA environment. The configuration is organised into multiple blocks, including node identity, service-specific ports and timeouts, file path references, and a reference to the primary ATM Server for initial registration and synchronisation.

4.6.5.1 ATM NODE DEFINITION

Listing 4.7 provides an example of identity and addressing profile of an ATC Agent within the ATMACA architecture. It specifies the agent's name, role (ATC_AGENT), realm, and the IP/host details used for establishing communication. This information is essential for registration, message routing, and address resolution across the network.

```
{
 "NodeId": 201,
 "NodeType": "AGENT",
 "NodeName": "ANKAREA",
 "NodeRole": "ATC_AGENT",
 "NodeRealm": "ankarea.atm",
 "NodeHost": "ankarea@global.atm",
 "NodeLocalAddress": "127.0.0.1",
 "NodeLocalPort": "DCL_DEFAULT_PORT"
}
```

Listing 4.7: Example an ATC Agent Node Configuration.

4.6.5.2 ATM-NODE-CONFIGURATION

Listing 4.8 illustrates the communication and operational behaviour of the Stationary ATC Client. It includes protocol-specific port assignments (e.g., CPDLC, DFIS, ADS), transport types, message timeout values, and parameters for peer connection monitoring. These settings govern how the client exchanges messages with agents, handles retries and enforces consistency in communication sessions. As the node is stationary, no mobility-triggered handovers are required, simplifying the runtime behaviour.

```
{ "NodeDlcmCommPort": 5910,
 "NodeDlcmTransportType": 2,
 "NodeCpdlcCommPort": 5811,
 "NodeCpdlcTransportType": 2,
 "NodeDFisCommPort": 5912,
 "NodeDFisTransportType": 2,
 "NodeAdsCommPort": 5913,
 "NodeAdsTransportType": 2,
 "NodeMsgTimeoutValue": 10000,
 "NodeMsgTimeoutCounter": 15,
 "NodeNumberofPeers": 100,
 "NodePeerConnAttemptCounter": 10,
 "NodePeerKeepAliveCounter": 10,
 "NodeNumberofFaultRecords": 100 }
```

Listing 4.8: Example of an ATC Stationary Client's behaviour.

4.6.5.3 ATM-NODE-PROVISION

Listing 4.9 lists file paths and operational datasets required for the node to function. It references configuration files, recovery files, logger settings, and paths to data elements such as flight information, protocol message catalogues, and global ATM Server references. These datasets are pre-provisioned by the Management Station and allow the client to start in a fully scoped operational state with no need for runtime redirection or reassignment.

```
{
 "NodeConfigurationFilePath": "../../data/NodeConfigurationFiles/",
 "NodeConfigurationFileName": "AtcAgentConfigurationFileANKAREA.json",
 "NodeRecoveryFilePath": "../../data/RecoveryFiles/AtcAgent/",
 "NodeRecoveryFileName": "NodeRecoveryFileANKAREA.json",
 "NodeLoggerFilePath": "../../data/LoggingFiles/AtcAgent/",
 "NodeLoggerFileName": "NodeLoggerFileANKAREA",
 "NodeDataFilePath": "../../data/DataFiles/AtcAgent/",
 "NodeCommonFilePath": "../../data/CommonFiles/",
 "NodeGlobalServerListFile": "AtmGlobalServerList.json",
 "NodeAtmCpdlcMessageListFile": "AtmCpdlcMessageList.json"
 "NodeApplicationList": ["DLCM"]
}
```

Listing 4.9: Example of an ATC Agent Node Provision.

4.6.5.4 ATM-SERVER-CONFIGURATION

Listing 4.10 specifies the identity and communication parameters of the upstream ATM Server to which the node connects for registration, provisioning synchronisation, and dataset updates. Although originally associated with Agent nodes, this block is also applicable to Stationary Clients, such as ATC Clients, which must establish a persistent link to the ATM Server to receive updates, carries out initial registration, or synchronise operational state with airspace data. In some deployments, the ATM Server information may be omitted from the main configuration block and instead retrieved from the Global Server List file (e.g., AtmGlobalServerList.json) referenced in the ATM-NODE-PROVISION section. This design supports more flexible ATM Server resolution based on geographic, operational, or policy-driven criteria.

```
{
 "AtmServerName": "EUROPE_REGION",
 "AtmServerRealm": "global.atm",
 "AtmServerHost": "EasternZone1@global.atm",
 "AtmServerLocalAddress": "192.168.1.29",
 "AtmServerLocalPort": "DCL_DEFAULT_PORT",
 "AtmServerLocalUDPPort": "DCL_DEFAULT_PORT"
}
```

Listing 4.10: Example ATC Agent Node ATM Server Configuration.

4.6.5.5 Context Management (CM) Agent

The CM Agent is a facility-scoped component within the ATMACA architecture, responsible for managing context state, including peer presence, user roles, and failover logic for clients associated with a specific operational site. Like the ATC Agent, the CM Agent is initialised using a JSON-based configuration file, which is transferred by the Management Station during deployment. This file defines the node's identity, communication parameters, runtime limits, and data provisioning paths.

The configuration content follows a structure similar to that of the ATC Agent, including node definition, communication port assignments, operational timeouts, and file system references. However, unlike ATC

Agents, which cover multiple facilities across an area, the CM Agent is tightly scoped to a single facility, and its configuration reflects this narrower operational context.

It also includes the identity of the upstream ATM Server (either directly or via the Global Server List), as well as references to provisioning datasets such as the Facility Table, Sector Table, and message catalogue files, which the agent will use to construct and manage user contexts. The CM Agent does not require knowledge of broader regional structure but instead focuses on maintaining consistency and availability within its assigned facility. This localised, lightweight design allows CM Agents to operate autonomously while remaining synchronised with the broader ATMACA system through their link to the ATM Server.

The CM Agent configuration file follows the same structural format as the ATC Agent's configuration file, consisting of node definition, communication parameters, provisioning paths, and upstream ATM Server information. The key difference lies in the assigned role, where the NodeRole is set to "CM_AGENT" to reflect the context management function of the node.

4.6.6 ATC Client

The ATC Client is a stationary client node within the ATMACA system, designed to support ground-based operational roles such as clearance delivery, flight monitoring, CPDLC session management, and mobility oversight. Although it may run on portable or mobile hardware, the ATC Client is considered stationary in architectural terms because its point of attachment to the ATC Agent remains fixed throughout its lifecycle. It does not carry out inter-agent handovers or context transfers.

The ATC Client is initialised using a structured JSON configuration file, provisioned by the Management Station prior to activation. In addition to standard fields such as NodeType (CLIENT) and NodeRole (STATIONARY_CLIENT), the configuration includes a NodeFunction field, which explicitly identifies the client's operational function: "ATC" in this case. This distinction enables finer classification of clients deployed within the same architecture, such as those used in tower operations, apron control, or airline dispatch.

The configuration also defines transport ports, timeout settings, and file system paths to essential datasets, including flight information, log files, and protocol message catalogues. While its structure is similar to that of a mobile client, the ATC Client operates with static agent association, allowing for simplified logic and persistent connectivity to its assigned ATC Agent and ATM Server. This ensures robust and consistent service delivery within facility-based environments such as control towers, ACCs, or airline operation centres.

4.6.6.1 ATM NODE DEFINITION

Listing 4.11 provides an example of the core identity and addressing profile of the Stationary ATC Client within the ATMACA architecture. Along with standard attributes such as NodeType (CLIENT) and NodeRole (STATIONARY_CLIENT), it introduces a new attribute (NodeFunction) to specify the functional purpose of the node in the operational environment.

```
{   "NodeId": 310,
  "NodeType": "CLIENT",
  "NodeName": "ESBTOWER01",
  "NodeRole": "STATIONARY_CLIENT",
  “Node Function”: ATC
  "NodeRealm": "esb.tr.atm",
  "NodeHost": "esbtower01@esb.tr.atm",
  "NodeLocalAddress": "127.0.0.1",
  "NodeLocalPort": "DCL_DEFAULT_PORT"   }
```

Listing 4.11: Example ATC Client/Stationary Node Configuration.

For example, a stationary client deployed in a control tower would have a NodeFunction value of "ATC", while other clients could be configured for "GROUND", "OCC", or "SUPPORT" roles. This additional field provides semantic clarity and improves service classification, especially when managing large networks of heterogeneous client nodes. As with other nodes, this section also specifies the NodeRealm, NodeHost, and NodeLocalAddress, ensuring unique identification and proper binding within the ATMACA domain.

4.6.6.2 ATM-NODE-CONFIGURATION

Listing 4.12 contains the ATC Agent's runtime communication parameters, including port assignments for various protocol services (e.g., DLCM, CPDLC, DFIS), transport type indicators (e.g., TCP/UDP), and operational policies such as timeouts, retries, and peer tracking. These settings govern how the agent interacts with its peers and maintains session resilience.

```
{
 "NodeDlcmCommPort": 5910,
 "NodeDlcmTransportType": 2,
 "NodeCpdlcCommPort": 5811,
 "NodeCpdlcTransportType": 2,
 "NodeDFisCommPort": 5912,
 "NodeDFisTransportType": 2,
 "NodeAdsCommPort": 5913,
 "NodeAdsTransportType": 2,
 "NodeMsgTimeoutValue": 10000,
 "NodeMsgTimeoutCounter": 15,
 "NodeNumberofPeers": 50,
 "NodePeerConnAttemptCounter": 10,
 "NodePeerKeepAliveCounter": 10,
 "NodeNumberofFaultRecords": 50
}
```

Listing 4.12: Example ATC Client/Stationary Node.

4.6.6.3 ATM-NODE-PROVISION

This section provides the file path definitions for configuration, recovery, logging, and operational datasets. It references the data files the ATC Agent uses at runtime, such as the global server list, CPDLC message list, and provisioning tables (**Listing 4.13**). These enable the agent to properly initialise, recover, and manage application-level data.

```
{
 "NodeConfigurationFilePath": "../../data/NodeConfigurationFiles/",
 "NodeConfigurationFileName": "StationaryClientConfig_ESBTOWER01.json",
 "NodeRecoveryFilePath": "../../data/RecoveryFiles/StationaryClient/",
 "NodeRecoveryFileName": "NodeRecoveryFile_ESBTOWER01.json",
 "NodeLoggerFilePath": "../../data/LoggingFiles/StationaryClient/",
 "NodeLoggerFileName": "NodeLoggerFile_ESBTOWER01",
 "NodeDataFilePath": "../../data/DataFiles/StationaryClient/",
 "NodeCommonFilePath": "../../data/CommonFiles/",
 "NodeFlightInfoFileName": "FlightInformation_ESBTOWER01.json",
 "NodeAtmGlobalServerListFile": "AtmGlobalServerList.json",
 "NodeAtmCpdlcMessageListFile": "AtmCpdlcMessageList.json"
 "NodeApplicationList": ["DLCM", "CPDLC", "DFIS", "ADS"]
}
```

Listing 4.13: Example ATC Client/Stationary Node Provision.

4.6.6.4 ATM-SERVER-CONFIGURATION

Listing 4.14 shows an example of the identity and communication parameters of the upstream ATM Server used by the node for registration, provisioning, and airspace dataset synchronisation. This block enables the agent to establish an initial connection with its designated ATM Server upon startup. In some deployments, this information may be omitted from the main configuration block and instead be retrieved from the Global Server List file (e.g., AtmGlobalServerList.json) referenced in the ATM-NODE-

PROVISION section. This allows more flexible resolution of ATM Server assignments based on geographic, operational, or policy-based routing.

```
{
 "AtmServerName": "EUROPE_REGION",
 "AtmServerRealm": "global.atm",
 "AtmServerHost": "EasternZone1@global.atm",
 "AtmServerLocalAddress": "192.168.1.29",
 "AtmServerLocalPort": "DCL_DEFAULT_PORT",
 "AtmServerLocalUDPPort": "DCL_DEFAULT_PORT"
}
```

Listing 4.14: Example ATC Client Node ATM Server Configuration.

4.6.7 Flight Deck Client

The Flight Deck Client is a mobile node deployed aboard aircraft and is responsible for maintaining active air-ground communications throughout all phases of flight. It supports critical airborne functions including CPDLC message exchanges, flight plan synchronisation, weather data access, and seamless context mobility across FIR and sector boundaries. Classified under the role MOBILE_CLIENT, this node actively switches ATC Agent as the aircraft transitions between control areas, requiring support for session continuity, context handover, and real-time rebinding.

The node is initialised using a structured JSON configuration file, provisioned by the Management Station before startup. In addition to the standard NodeType and NodeRole attributes, the configuration includes a NodeFunction field, set to "FLIGHTDECK", which explicitly defines the operational purpose of the client within the ATMACA system. This term was intentionally chosen over alternatives such as "Pilot Client" or "Aircraft Client" to strike a balance between human-centric operations (e.g., pilot-controlled messaging) and future autonomy, where the flight deck may operate under machine-driven logic without human intervention. The term "Flight Deck" remains relevant across this evolution, as it refers to the system's operational domain, not just its human operator, ensuring long-term compatibility with both manned and unmanned aircraft deployments.

The configuration also includes service port definitions, timeouts, provisioning paths, and ATM Server details to support dynamic registration, message routing, and mobility-aware service binding. This architecture enables the Flight Deck Client to maintain resilient communication services and contextual integrity across all network boundaries encountered in flight.

4.6.7.1 ATM NODE DEFINITION

This section defines the identity and addressing profile of the Flight Deck Client (**Listing 4.15**). It includes key attributes such as the NodeType (CLIENT), NodeRole (MOBILE_CLIENT), and NodeFunction (FLIGHTDECK), which together characterise the node as an aircraft-based mobile entity responsible for airborne communication and context management. The use of "FLIGHTDECK" as the function ensures compatibility with both current human-operated systems and future autonomous environments, anchoring the node to its operational domain rather than its user type. Additional fields such as NodeHost, NodeRealm, and NodeLocalAddress support message routing and network-level discovery.

```
{  "NodeId": 320,
  "NodeType": "CLIENT",
  "NodeName": "TCJFK123",
  "NodeRole": "MOBILE_CLIENT",
  "NodeFunction": "FLIGHTDECK",
  "NodeRealm": "air.tr.atm",
  "NodeHost": "tcjfk123@air.tr.atm",
  "NodeLocalAddress": "127.0.0.1",
  "NodeLocalPort": "DCL_DEFAULT_PORT" }
```

Listing 4.15: Example ATC Client/Stationary Node Configuration.

4.6.7.2 ATM-NODE-CONFIGURATION

Listing 4.16 defines the transport and session handling behaviour of the Flight Deck Client. It includes port assignments for DLCM, CPDLC, ADS, and DFIS services, as well as protocol transport types (e.g., TCP/UDP), message timeout settings, and parameters related to peer connectivity and fault tracking. These settings are critical for ensuring session resilience and responsiveness as the mobile node transitions between ATC Agents and traverses airspace boundaries. They allow the client to manage connection lifecycles effectively under highly dynamic operational conditions.

```
{
 "NodeDlcmCommPort": 5910,
 "NodeDlcmTransportType": 2,
 "NodeCpdlcCommPort": 5811,
 "NodeCpdlcTransportType": 2,
 "NodeDFisCommPort": 5912,
 "NodeDFisTransportType": 2,
 "NodeAdsCommPort": 5913,
 "NodeAdsTransportType": 2,
 "NodeMsgTimeoutValue": 10000,
 "NodeMsgTimeoutCounter": 15,
 "NodeNumberofPeers": 100,
 "NodePeerConnAttemptCounter": 10,
 "NodePeerKeepAliveCounter": 10,
 "NodeNumberofFaultRecords": 50
}
```

Listing 4.16: Example ATC Client/Stationary Node Configuration.

4.6.7.3 ATM-NODE-PROVISION

This section provides the file system structure and dataset references needed for the client to operate (**Listing 4.17**). It includes file paths for configuration files, logging directories, recovery states, and domain-specific data such as flight information and protocol message catalogues. These resources are provisioned by the Management Station prior to startup and allow the client to initialise in a self-contained, context-aware state. For mobile clients, the data path also supports dynamic updates as the flight transitions across regions.

```
{
 "NodeConfigurationFilePath": "../../data/NodeConfigurationFiles/",
 "NodeConfigurationFileName": "FlightDeckClientConfig_TCJFK123.json",
 "NodeRecoveryFilePath": "../../data/RecoveryFiles/FlightDeckClient/",
 "NodeRecoveryFileName": "NodeRecoveryFile_TCJFK123.json",
 "NodeLoggerFilePath": "../../data/LoggingFiles/FlightDeckClient/",
 "NodeLoggerFileName": "NodeLoggerFile_TCJFK123",
 "NodeDataFilePath": "../../data/DataFiles/FlightDeckClient/",
 "NodeCommonFilePath": "../../data/CommonFiles/",
 "NodeFlightInfoFileName": "FlightInformation_TCJFK123.json",
 "NodeAtmGlobalServerListFile": "AtmGlobalServerList.json",
 "NodeAtmCpdlcMessageListFile": "AtmCpdlcMessageList.json"
 "NodeApplicationList": ["DLCM", "CPDLC", "DFIS", "ADS"]
}
```

Listing 4.17: Example ATC Client/Stationary Node Provision.

4.6.7.4 ATM-SERVER-CONFIGURATION

Unlike stationary clients and agent nodes that register with a fixed ATM Server, mobile clients such as the Flight Deck Client do not include a static ATM-SERVER-CONFIGURATION block. This is because the client is expected to operate across multiple FIRs, ATC regions, and network domains during flight.

Instead, mobile clients are provisioned with a Global Server List (e.g., AtmGlobalServerList.json) referenced in the ATM-NODE-PROVISION block. This list enables the client to dynamically discover and connect to the appropriate ATM Server based on location, domain preference, or operational policy. This decoupled design supports true global mobility while maintaining registration flexibility and failover resilience.

4.6.8 Application Server

The Application Server is a SERVER-type node within the ATMACA architecture that hosts and delivers higher-layer operational services such as DFIS and GRO. These nodes are typically deployed at the area or regional level, serving as centralised providers of digital aeronautical content to ATC Agents, CM Agents, and client nodes. The node is initialised using a structured JSON configuration file, provisioned by the Management Station during deployment. Two key fields control its behaviour:

The NodeApplicationList specifies which internal service modules should be activated at runtime, ensuring modular deployment and flexibility.

The NodeServiceList declares which externally advertised services the Application Server makes available to the broader ATMACA network. These may include DFIS such as weather services and Notice to Airmen (NOTAM) services, or other custom services relevant to operational or regional needs.

By separating internal application logic from externally discoverable services, the configuration supports dynamic service discovery, scalable provisioning, and policy-based exposure of capabilities. This structure allows a single server platform to host multiple services while exposing only those required in a given operational scenario.

4.6.8.1 ATM NODE DEFINITION

The ATM-NODE-DEFINITION section defines the identity, addressing, and role of the Application Server within the ATMACA network (**Listing 4.18**). As a SERVER-type node with the role of "APPLICATION_SERVER", it is uniquely named and associated with a logical realm (e.g., "dfis-server@east.dfis.tr.atm"). This information ensures the node can be discovered, authenticated, and addressed correctly during registration and service discovery. The NodeHost field serves as the unique reference for inter-node communication, and the NodeLocalAddress identifies the network interface the server binds to during operation.

```
{
 "NodeId": 410,
 "NodeType": "SERVER",
 "NodeName": "DFIS_APP_SERVER_EUR",
 "NodeRole": "APPLICATION_SERVER",
 "NodeFunction": "DFIS",
 "NodeRealm": "east.dfis.tr.atm",
 "NodeHost": "dfis-server@east.dfis.tr.atm",
 "NodeLocalAddress": "127.0.0.1",
 "NodeLocalPort": "DCL_DEFAULT_PORT"
}
```

Listing 4.18: Example Application Server Node Configuration.

4.6.8.2 ATM-NODE-CONFIGURATION

The ATM-NODE-CONFIGURATION block defines the operational parameters and communication settings for the Application Server (**Listing 4.19**). This includes port numbers for application-layer communication (e.g., DLCM, DFIS), transport protocol types (e.g., TCP or UDP), message timeout settings, and peer management behaviours. These values ensure that the Application Server participates in reliable message exchange, supports concurrent peer connections, and enforces communication rules consistent with ATMACA's runtime architecture.

While Application Servers are not expected to engage in high-frequency peer interactions like agents, they must remain responsive and available for ATC Agent-mediated service requests. These settings define the behaviour of the server under load, during retries, and when recovering from faults.

```
{
 "NodeDlcmCommPort": 5910,
 "NodeDlcmTransportType": 2,
 "NodeDfisCommPort": 5921,
 "NodeDfisTransportType": 2,
 "NodeMsgTimeoutValue": 10000,
 "NodeMsgTimeoutCounter": 15,
 "NodeNumberofPeers": 20,
 "NodePeerConnAttemptCounter": 5,
 "NodePeerKeepAliveCounter": 5,
 "NodeNumberofFaultRecords": 50
}
```

Listing 4.19: Example Application Server Node Configuration.

4.6.8.3 ATM-NODE-PROVISION

The ATM-NODE-PROVISION section defines the file paths and runtime resource references required by the Application Server (**Listing 4.20**). These include paths to configuration files, log files, recovery data, and domain-specific service datasets such as NOTAM messages, weather feeds, and airport information. It also includes the NodeApplicationList and NodeServiceList, which determine the internal modules that should be activated and the external services that will be advertised to the ATMACA network.

This provisioning data is transferred by the Management Station during the initialisation phase and enables the Application Server to boot with complete operational context. The modular design allows different application servers to be adapted to their roles (e.g., NOTAM-only, weather-only, or combined DFIS).

```
{
 "NodeConfigurationFilePath": "../../data/NodeConfigurationFiles/",
 "NodeConfigurationFileName": "DfisServerConfig_EAST.json",
 "NodeRecoveryFilePath": "../../data/RecoveryFiles/ApplicationServer/",
 "NodeRecoveryFileName": "NodeRecoveryFile_DFIS.json",
 "NodeLoggerFilePath": "../../data/LoggingFiles/ApplicationServer/",
 "NodeLoggerFileName": "NodeLoggerFile_DFIS",
 "NodeDataFilePath": "../../data/DataFiles/ApplicationServer/",
 "NodeCommonFilePath": "../../data/CommonFiles/",
 "NodeAtmGlobalServerListFile": "AtmGlobalServerList.json"

 "NodeApplicationList": ["DLCM"],
 "NodeServiceList": ["NOTAM", "WEATHER", "AIRPORT_INFO"]
}
```

Listing 4.20: Example Application Server Node Provision.

4.6.8.4 ATM-SERVER-CONFIGURATION

The ATM-SERVER-CONFIGURATION block specifies the identity and communication parameters of the upstream ATM Server that the Application Server may register with for provisioning, monitoring, or coordination (**Listing 4.21**). This configuration is optional for Application Servers, as some deployments prefer dynamic resolution of ATM Servers using a Global Server List (e.g., AtmGlobalServerList.json), which is referenced in the ATM-NODE-PROVISION section.

Including this section directly allows for static binding to a designated ATM Server during startup, which is useful in fixed regional deployments or constrained operational environments. Alternatively, using a global server list supports more dynamic and scalable discovery of ATM services based on geography, redundancy, or routing policy.

```
{
  "AtmServerName": "ATM-SERVER-TR-EAST",
  "AtmServerRealm": "east.tr.atm",
  "AtmServerHost": "atm-east@east.tr.atm",
  "AtmServerLocalAddress": "192.168.10.10",
  "AtmServerLocalPort": "DCL_DEFAULT_PORT",
  "AtmServerLocalUDPPort": "DCL_DEFAULT_PORT"
}
```

Listing 4.21: Example Application Server Configuration.

4.6.9 Application Initialisation during Node Startup

In the ATMACA protocol stack, application initialisation is the process by which each node activates the protocol modules, services, and runtime components required to fulfil its operational role. This process is initiated during node startup, and its behaviour is driven by a combination of the node type, role, and the explicitly defined NodeApplicationList in the configuration file.

Each application listed in NodeApplicationList corresponds to a protocol-level or service-layer module such as DLCM (for connection/session/context handling), GRO, or DFIS like NOTAM or Weather. These modules are selectively initialised based on the node's operational responsibilities, ensuring a modular and efficient runtime environment. Initialisation typically involves:

1. Loading message dictionaries for protocol parsing
2. Registering application-level callbacks (e.g., message handlers)
3. Launching service threads or event handlers
4. Binding to transport-layer sockets (based on port settings)

This modular startup process allows nodes with the same architecture to be configured differently based on mission or deployment scope and ensures scalability, flexibility, and fault isolation within the ATMACA ecosystem. While application initialisation behaviour in ATMACA is adapted to each node type, the underlying process follows a consistent and modular framework implemented in the system runtime. This ensures uniform startup handling across all nodes, whether they are CLIENT, AGENT, or SERVER types.

Initialisation begins within the node's startup logic typically using the AtmNode::Run() or DlcNetwork::Init() routines and is driven by function mappings associated with the node's declared type.

- **Initialisation Function Mapping**: The system uses a centralised dispatch table (atmNodeInitFuncRetriever) to associate each AtmNodeType with its corresponding startup function such as atm_client_init, atm_agent_init, or atm_server_init.
- **Node State Check**: Initialisation proceeds only if the node is in NODE_SETUP state, ensuring proper sequencing and preventing reinitialisation during failure recovery.
- **Role-Specific Initialisation:** (example: atm_client_init())
  - Loads flight, sector, or facility configuration.
  - Retrieves the ATC Agent association from a global configuration file.
  - Sets application flags (e.g., FLAGS_DLCM, FLAGS_CPDLC) based on NodeApplicationList.
  - Calls atm_sl_init() and atm_sl_start() with those flags to bring up relevant service layers.
- **Callback Registration**: Via atm_service_layer_callback_setting(flags), message handlers such as ProcessDlcmRequest() and ProcessCpdlcRequest() are linked to the dispatcher for handling protocol messages.
- **Message Dictionary and Worker Threads**:
  - Dictionaries are loaded using functions like add_cpdlc_message_to_dictionary_of_this_node().
  - Messages are routed to worker threads via POST_MSG_TO_THREAD() for asynchronous processing.
- **Recovery and Peer Initialisation**:
  - Optionally reads a recovery file to restore session or peer state.
  - Establishes outbound peer connections (e.g., to an ATC Agent or ATM Server), validating handshake and service readiness.

4.6.10 Node Recovery and State Management

ATMACA nodes implement a structured state management and recovery mechanism to ensure stability, fault tolerance, and continuity of operations across diverse deployment environments. Each node transitions through a well-defined lifecycle, and in the event of an unexpected shutdown, crash, or loss of connectivity, it can reinitialise its operational context using the node's recovery file. This mechanism enables seamless reattachment to the network, resumption of critical sessions, and minimal disruption.

State and recovery handling is managed by the core node logic and may vary slightly by node type (e.g., SERVER, AGENT, CLIENT), but it always adheres to the following principles:

- **State-Aware Initialisation**: Nodes validate their internal state before transitioning to active operation.
- **Context-Aware Recovery**: Previously stored runtime data (e.g., peer lists, session bindings) are reloaded to resume operation with minimal loss.
- **Graceful Degradation**: Partial functionality is preserved if full recovery is not possible.
- **Resumable Sessions and Peer Associations**: Ongoing sessions and peer roles are restored to maintain protocol continuity.

4.6.10.1 Node State Lifecycle

Every ATMACA node operates within a structured state lifecycle that governs its readiness, activity, and response to failure or shutdown events. **Table 4.2** illustrates this lifecycle which provides a controlled framework for managing initialisation, operational execution, error handling, and graceful recovery. The Node Manager module uses this lifecycle to enforce proper sequencing, prevent unsafe transitions, and ensure resilience during runtime fluctuations.

Table 4.2: Primary Node States in the ATMACA Lifecycle.

State	Description
UNKNOWN	Default state before any initialisation begins: The node has not yet been configured.
NODE_SETUP	Configuration has been loaded, but subsystems and services have not yet been initialised.
NODE_INIT	Services, internal modules, and runtime flags are initialised: The node prepares for operation.
NODE_START	The node enters active service mode, begins peer interaction, handles protocol traffic.
NODE_STOP	A temporary or soft halt is requested: Services stop gracefully, but the process is not yet exited.
NODE_SHUTDOWN	Final shutdown in progress: resources are released, and persistent state is saved if enabled.

The lifecycle flow is as follows:

1. Startup begins in the UNKNOWN state.
2. When configuration is successfully parsed, the node enters NODE_SETUP.
3. Once the initialisation routine (atm_node_init()) is invoked, it transitions to NODE_INIT, where message handlers, applications, and transport bindings are registered.
4. The transition to NODE_START signals that the node is live and serving its operational role.
5. Administrative requests or faults may move the node to NODE_STOP for graceful temporary deactivation.
6. Finally, NODE_SHUTDOWN completes resource cleanup and exits the process.

This model allows a lightweight but robust control mechanism for managing node behaviour across the full lifecycle, ensuring that ATMACA components initialise safely, operate efficiently, and shut down gracefully with support for both manual and automated state transitions.

4.6.10.2 Node Recovery Mechanism

ATMACA implements a robust, restart-oriented node recovery mechanism designed to maintain operational continuity in the face of unexpected disruptions including software crashes, power loss, or controlled reboots. Central to this capability is a structured recovery process that uses role-specific recovery logic, checkpointed node state, and pre-defined recovery files. Upon startup, each node whether Server, Agent, or Client, examines its environment for an existing recovery file, parses its contents, and determines whether to resume previous sessions, reattach to peers, or reinitialise from a clean state. This design minimises service interruption, preserves session continuity, and enables seamless restoration of critical runtime context. Recovery is particularly vital in mobility-driven scenarios, session-centric communication flows, and systems requiring high availability under dynamic operational conditions.

Node Recovery File Operations

Node Recovery File Operations in ATMACA are central to enabling consistent and fault-resilient restarts across all node types. Upon initialisation, each node inspects a designated recovery file, typically stored locally in a predefined path, containing serialised information about its previous runtime state. This file captures critical metadata such as peer associations, session bindings, node role and type, and application flags. If a valid recovery file is found, the node uses its contents to selectively rehydrate its internal structures, reestablish communication with previously connected peers, and restore context or session continuity. This mechanism minimises the operational impact of transient outages by avoiding full reinitialisation and allows faster reintegration into the ATMACA network.
The functions are as follows:

- ReadNodeRecoveryFile(): Reads the contents of the node recovery file.
- UpdateNodeRecoveryFile(): Updates the node recovery file with current node status or data.
- DeleteNodeRecoveryFile(): Deletes the recovery file when it is no longer needed.
- CreateNodeRecoveryFile(string fileName): Creates a recovery file with the specified name.

Node Restart Mechanism

ATMACA supports multiple recovery methods tailored to the operational needs and severity of the disruption experienced by a node. These methods, classified as Hot Restart, Warm Restart, and Cold Restart, dictate how much of the previous runtime state is restored upon reboot and how the node reintegrates into the ATMACA network (**Table 4.3**). Each recovery type balances trade-offs between continuity, safety, and initialisation overhead. The selected method influences whether existing sessions are resumed, peers are reconnected automatically, or a complete reinitialisation is carried out. This tiered recovery strategy provides flexible fault tolerance, allowing nodes to adapt recovery behavior to the nature of the failure, the node's role, and system-wide policy constraints.

The recovery strategy is guided by the NodeRestartType enumeration, which determines the level and scope of recovery actions to be taken during reinitialisation.

Table 4.3: Node Restart based on the Type.

Restart Type	Description
NONE	No recovery is carried out. The node starts cleaning without restoring any previous state. Used for fresh deployments.
HOT_RESTART	Restores full runtime state, including active sessions, peer associations, and timers. No application reinitialisation is required. Ideal for quick reboots where memory and runtime context are preserved.
WARM_RESTART	Restores partial runtime state (e.g., session metadata, peer roles), but reinitialises applications and message handlers. Balances speed and safety.
COLD_RESTART	Reinitialises all internal components and loads configuration from scratch. Only static files and provisioning data are reused. This is the most comprehensive recovery type and is equivalent to a full restart.

4.6.11 Logical (Operational) Entities – Node Association and Management

ATMACA defines a structured relationship between physical nodes (e.g., ATC Agents, CM Agents, Clients, and Servers) and logical representations of airspace structures, such as areas, facilities, sectors, and flights. These logical entities represent the operational dimensions of the ATMACA architecture, and every node, regardless of its physical deployment, is associated with one or more of these elements to establish context, scope, and function.

Each ATC Agent is mapped to an area, which in turn encompasses multiple facilities, each containing one or more sectors. The ATC Agent maintains full awareness of this hierarchical structure, including adjacent areas, enabling the system to coordinate handovers, maintain session continuity, and manage dynamic peer relationships. Through this structure, the ATC Agent not only handles communication at the transport level but also acts as an operational proxy for the sectors and facilities under its control.

CM Agents are scoped more narrowly, and they are associated with a single facility and are responsible for tracking user contexts, presence, and roles within that facility's sectors. This assignment allows CM Agents to carry out fine-grained context management and maintain real-time awareness of user mobility within their operational boundaries.

On the client side, stationary ATC Clients are directly associated with specific sectors, representing operational points of control such as tower or en-route positions. Meanwhile, Flight Deck Clients are associated with flights as logical entities and maintain dynamic relationships with sectors and areas as they traverse across FIRs. These relationships form the backbone of airborne mobility and session continuity.

Though these mappings represent logical associations, they are not merely static references. Each ATMACA node, via its software components, establishes runtime awareness and interaction with its associated logical entities. This association enables the node to process context-specific messages, participate in mobility operations, and enforce access policies based on its operational domain.

This logical-to-physical binding is managed and orchestrated through software constructs such as the AtmLogicalNodeInfo and AtmLogicalNodeManager, which store, retrieve, and expose these relationships during runtime. These components allow the ATMACA software stack to maintain an abstract yet operationally relevant view of airspace entities enabling modular deployments, policy-based routing, and seamless coordination across the distributed network.

The current implementation of logical node association and management is encapsulated within the AtmLogicalNodeInfo and AtmLogicalNodeManager modules. These components are responsible for mapping physical ATMACA nodes (e.g., ATC Agents, CM Agents, Clients) to their corresponding logical entities such as areas, facilities, sectors, and flights. However, the naming of these modules is provisional and may be subject to change to better reflect their architectural role or to align with broader naming conventions adopted across the system. Any such renaming will preserve the underlying logic and data model while improving clarity and maintainability for future developers and integrators.

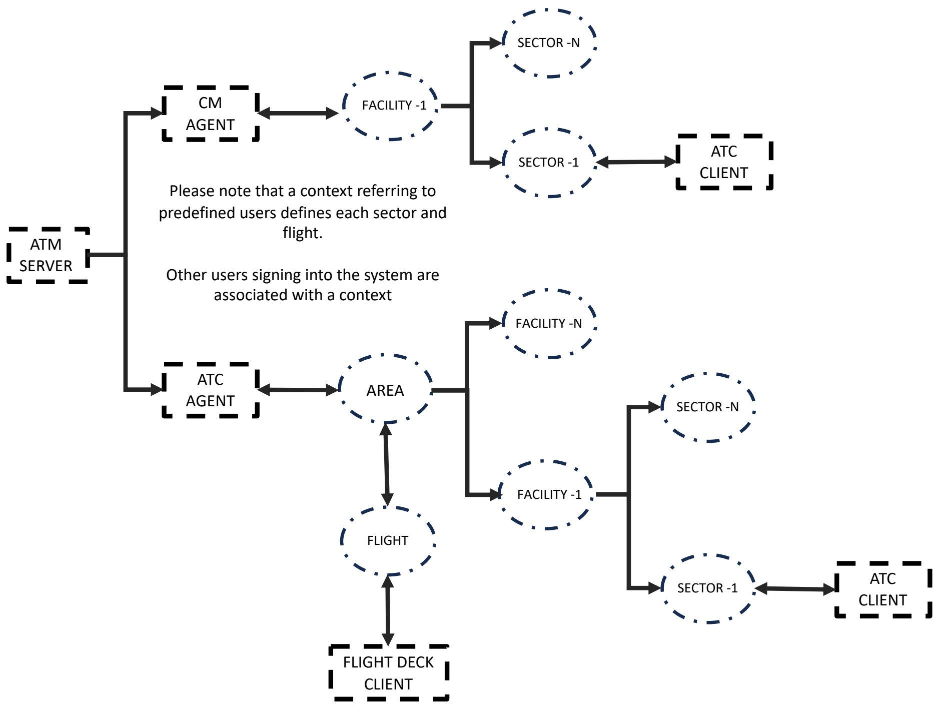

SECTOR -N
CM AGENT
FACILITY -1
SECTOR -1
ATC CLIENT
Please note that a context referring to predefined users defines each sector and flight.
Other users signing into the system are associated with a context
ATM SERVER
FACILITY -N
ATC AGENT
AREA
SECTOR -N
FACILITY -1
FLIGHT
SECTOR -1
ATC CLIENT
FLIGHT DECK CLIENT

Figure 4.11: Logical (Operational) Entities – Node Association in ATMACA Network.

Figure 4.11 represents the communication topology in the ATMACA system, showing how the ATM Server connects through CM and ATC Agents to relay data and control flows toward various ATC Clients and the Flight Deck Client via intermediary nodes.

4.6.12 Protocol Level Connection Management

In the ATMACA protocol stack, peer connection management is a core responsibility of the AtmPeerManager and its associated peer and peer group abstractions. The system adopts a modular and extensible model for defining, creating, managing, and recovering peer connections across all node types, including agents, clients, and servers. These connections form the basis for secure, session-resilient, and failover-tolerant communication between distributed nodes in the ATMACA architecture.

4.6.12.1 Architecture Overview

Peer management is layered into three key components:

- PeerManager: Orchestrates the lifecycle of peer entities.
- Peer: Represents a connection to a remote node, encapsulating its state, socket, watchdog, and queue mechanisms.
- PeerGroup: Provides a logical grouping of peers, enabling failover and message rerouting.

All these components use callback-driven state monitoring and transaction-level inspection to maintain high availability and responsiveness across air-ground and inter-agent connections.

4.6.12.2 Peer Initialisation and Discovery

Peers can be configured via:

- Static definitions in configuration files (atmc://...;name=peer1;type=AGENT;role=ATC_AGENT).
- Dynamic creation during runtime via APIs (e.g., CreatePeer(...)).
- Auto-discovery on connection attempts using IP and port matching logic governed by system policies like enable-IP-matching.

If a peer is not found upon an incoming connection, and dynamic peer creation is enabled, the peer manager auto-creates a new AtmacaPeer instance and informs the upper layer for possible validation and use.

4.6.12.3 Connection Models and Modes

The connection modes define how a peer operates in maintenance scenarios. These modes govern how a peer behaves in terms of initiating, accepting, or withholding connection attempts.

Table 4.4: Peers support three operational connection modes.

Maintenance Mode	Description
active	Peer initiates connection to a known remote endpoint.
passive	Waits for incoming connection requests. Typical of server-mode behaviour.
offline	No connection is attempted; peer exists for monitoring or reference only.

For active peers, connection attempts continue based on reconnect-timeout. If a connect-timeout elapses, the application is notified, and it can decide whether to stop retrying (**Table 4.4**).

4.6.12.4 Callback Interfaces and Event Handlers

The system provides rich callback interfaces across:

- Peer Callbacks: OnConnect, OnDisconnect, OnConnectFailure, OnRemoteError, etc.
- PeerGroup Callbacks: Handle failover events, disconnects, and reassignments.
- Message Observers: Monitor state changes, errors, and timeouts of protocol messages.
- Handlers: Act as primary processors of received request/response messages.

These interfaces ensure that application logic remains aware of and responsive to runtime events such as message failures, connection interruptions, or recoveries.

4.6.12.5 Peer Groups and Failover

Peers can be grouped into PeerGroups, enabling redundancy and load-balancing:

- When a primary peer fails, queued messages are re-routed to the next available peer.
- Failover decisions are governed via peerGroupCallbackOnDisconnect and message enumeration logic.
- If messages cannot be rerouted, they are detached for manual inspection and reprocessing.

This mechanism aligns with RFC 6733-inspired failover and failback principles and ensures minimal disruption under dynamic operational conditions [5].

4.6.12.6 Watchdog and Health Monitoring

ATMACA's connection reliability is reinforced via a Watchdog State Machine, implemented per RFC 3539. Peers periodically exchange Device-Watchdog-Request and Response messages:

- If a peer fails to respond within the configured timeout, it is considered disconnected.
- Callback peerCallbackOnDisconnect is invoked, and reconnection attempts begin.

4.6.12.7 Runtime Socket and Transport Abstraction

Each peer supports transport configuration including:

- Protocol (TCP/UDP/SCTP)
- Local and remote IP binding
- Multiple IP fallback using round-robin attempt order

This allows peer definitions to support multi-homing and dynamic network routing strategies common in ATC networks and airborne systems.

4.6.12.8 Dynamic Message Handling and Recovery

ATMACA peers can reroute, cancel, or reassign pending transactions based on connection state:

- Messages are stored in orphan queues during failover.
- Application callbacks inspect and redirect or cancel based on peer availability.
- This mechanism ensures that in-flight messages are never lost silently.

This peer management architecture reflects a robust, event-driven, and fault-resilient design adapted to the dynamic, distributed nature of air traffic and mobility-driven protocol environments.

4.6.13 Routing and Forwarding Table Construction

To support efficient and context-aware message delivery, each ATMACA node constructs and maintains a Routing and Forwarding Table (RFT) as part of its startup and operational lifecycle. This table maps logical operational entities such as sectors, facilities, or flights, to physical or virtual peer endpoints based on their association with areas, roles, and declared service capabilities. The table is constructed using information derived from the node configuration, dynamic provisioning (e.g., sector and area adjacency data), and peer registration state.

The RFT serves as a central lookup structure for session routing, mobility-driven rebinding, and service endpoint discovery. For example, when a mobile client transitions across areas or sectors, the RFT allows the node to quickly determine the correct ATC Agent or Application Server to reroute communication. It also supports multi-tiered fallback logic by associating services with primary and backup routes based on peer availability.

Peer connection handling and routing logic are embedded in a tightly coupled set of classes, namely the AtmPeerManager (for maintaining active peer relationships) and RoutingTableElementData (for managing route decision-making based on logical addressing).

This section introduces the data structures and algorithms used to build and maintain the RFT, explains how routing logic adapts to mobility and operational changes, and highlights its role in ensuring low-latency, policy-compliant message delivery in ATMACA's distributed runtime environment.

4.6.13.1 Routing Table as Integrated Forwarding and Routing Structure

The RoutingTableElementData class serves dual purposes, routing decisions and message forwarding, by maintaining a central map of destination-to-next-hop associations. The routing table is implemented as a singleton instance, globally accessible through the macro-ATM_ROUTEDATA_TABLE(). Each route entry includes:

- Entity Identifier: Logical identifier (e.g., realm or hostname) used for address resolution.
- Adjacent Node: The next-hop node (e.g., ATC Agent or facility agent).
- Application ID: Identifies the relevant application service (default is 0 for generic routes).
- Routing Action: Enum indicating the behaviour (e.g., LOCAL, RELAY, or REDIRECT).
- Dynamic Creation Flag: Indicates whether the route is static or generated at runtime.
- Time to Live (TTL): Differentiates between persistent static routes (STATIC_ROUTE_TTL) and temporary dynamic entries (DYNAMIC_ROUTE_TTL).

4.6.13.2 Peer Discovery and Route Creation

During node operation or registration, routing entries may be created dynamically based on incoming messages. For instance:

- Facility-Based Routing: Triggered by parsing incoming facility messages and associating them with known ATC Agents.
- Flight-Based Routing: Established by extracting origin information from mobile client messages (e.g., originating aircraft) and binding it to its currently serving ATC Agent.

These methods are implemented through helper functions where each helper reads the origin host and agent node name from message DIX structures and updates the route table accordingly.

4.6.13.3 Routing Actions

The system uses the AtmRoutingAction enum to determine how messages should be processed:

- LOCAL: Message is consumed by the current node.
- RELAY: Message should be forwarded to the next-hop peer (typically for mobile clients or sector-based forwarding).
- REDIRECT: Message is rerouted due to policy or network constraints (e.g., agent reassignment).

This action type is embedded in every route entry and governs how the Node Manager and transport layer treat outbound messages.

4.6.13.4 Routing Table Visualisation and Debugging

The system provides built-in mechanisms for inspecting and validating the routing state at runtime:

- view_routing_table() prints all entries, differentiating between static and dynamic routes.
- retrieve_route_data_from_routing_table() enables lookup of routing metadata by entity identifier.
- delete_routing_table() flushes all entries, used during shutdown or configuration reloads.

Such runtime visibility is crucial for debugging live systems, handling failovers, and confirming dynamic route behavior.

4.6.13.5 Integration with Peer Manager

While RoutingTableElementData governs routing logic, actual peer connection lifecycles are managed by the AtmPeerManager, which creates, validates, and monitors peer links. When a route is created or selected for use, the corresponding peer connection is resolved and activated. Peer management also tracks:

- Handshake status
- Keepalive supervision
- Fault recovery and retry intervals
- Peer readiness for message forwarding

Route resolution and peer tracking work in concert to ensure robust protocol delivery, even under dynamic topology changes.

4.6.14 Node Management Interface

The AtmMgmtInterface class provides a static, menu-driven interface for managing ATMACA nodes from the command line. It enables users to define, create, update, and delete ATM nodes, as well as view and modify their configuration files. This interface plays a vital role in setting up and maintaining node-specific settings in both development and operational environments.

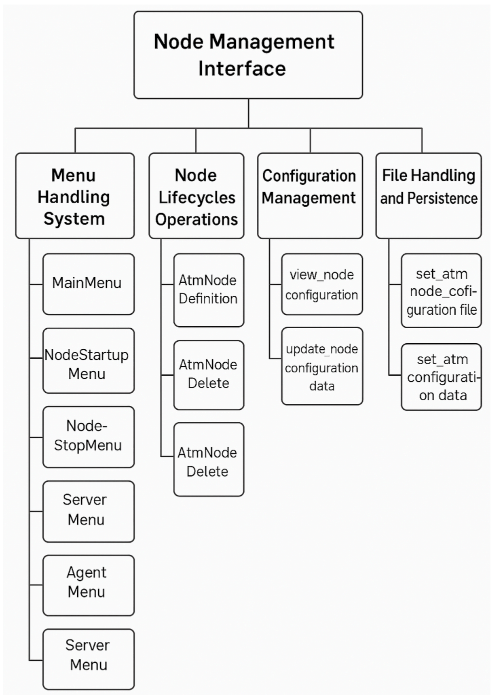


Figure 4.12: Structured View of the Node Management Interface.

Figure 4.12 shows the structure of the ATMACA node management interface, detailing its main functional areas including menu navigation, node lifecycle operations, configuration management, and file handling for system persistence.

4.6.14.1 Menu Handling System

The interface uses a static array of function pointers, menuFunctions[MAXMENUFUNC], to map menu choices to the corresponding handler methods. It supports various menu layers, with methods like MainMenu, NodeStartupMenu, NodeRestartMenu, and NodeStopMenu allowing users to navigate through operational tasks. Specialised menus also exist for roles such as server, agent, client, and different ATM modules like Departure Clearance (DLC), DLR, ATC Communication Management (ACM), and ATC Clearance (ACL). Each menu function receives user input and calls for the appropriate routine, providing modular and user-friendly navigation.

4.6.14.2 Node Lifecycle Operations

The interface allows for basic lifecycle control of ATM nodes through static methods like AtmNodeCreate, AtmNodeDelete, and AtmNodeUpdate. These operations manipulate node metadata or configuration structures and can operate interactively or be triggered by menu options. The method AtmNodeDefinition helps initialise node parameters and is commonly used during setup workflows.

4.6.14.3 Configuration Management

Node configuration can be viewed using the overloaded method view_node_configuration, which either takes an AtmNode* pointer or retrieves the current context. Updates to configuration data are supported via update_node_configuration_data, again with overloaded signatures. This dual design ensures flexibility for both script-based and interactive usage.

For fine-grained edits, developers or operators can use set_individual_node_conf_parameters, which lets users change specific fields in the configuration. Parameters are identified by integer IDs, aligning with structured definitions elsewhere in the stack.

4.6.14.4 File Handling and Persistence

The interface provides methods to handle file names and paths dynamically, such as set_atm_node_configuration_filename and set_atm_node_configuration_filepath. This enables flexible storage, loading, and reuse of node configuration files across environments. To ensure portability and transparency, the system supports saving configuration files in JSON format using save_configuration_data_in_json_format, available with or without an AtmNode context.

4.6.14.5 Initialisation Utilities

To bootstrap the interface, initialize_startup_menu sets up the default menu function bindings. Then, create_startup_menu is used to render and manage a startup UI experience based on the current menu level and context. These functions are crucial during system bootstrapping or when operating in manual recovery or fallback modes.

5 CONCLUSION

This report described the implementation architecture of the ATMACA communication protocol, presenting the specification of its key components including the ATM and application servers, clients, and communication agents. It provided a detailed description of how provisioning, session handling, and context-based communication are structured to support dynamic and scalable air traffic management. The next phase will involve the implementation and integration of the DLCM application in deliverable D4.3 followed by CPDLC-DLCM integration and system-level testing to validate interoperability, performance, and compliance with the requirements defined in deliverable D3.1 [7]. Appendix A includes a table outlining the requirements mapping, indicating what is currently implemented and what will be addressed by the DLCM application.

6 References

6.1 Applicable documents

Content integration

[1] Common Taxonomy, ed. 01.00, 07/02/2023

[2] Content Integration – Executive Overview, ed. 00.01, 16/02/2023

Project and programme management

[3] 101167070 ATMACA Grant Agreement, 25/06/2024

[4] SESAR 3 JU Project Handbook – Programme Execution Framework

6.2 Reference documents

[5] V. Fajardo, J. Arkko, J. Loughney, and G. Zorn, "Diameter Base Protocol," RFC 6733, IETF, Oct. 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6733

[6] D. Crocker and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," RFC 5234, IETF, January 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5234.html

[7] ATMACA-SESAR, Deliverable D3.1, ATMACA Protocol Design Plan, 2025

[8] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261, Jun. 2002

[9] ICAO Doc 9880, Manual on Detailed Technical Specifications for the Aeronautical Telecommunication Network (ATN) using ISO/OSI Standards and Protocols, First Edition - 2010

[10] ICAO Doc 9896, Manual for the ATN using IPS Standards and Protocols, 1st edition

[11] ICAO Doc 4444, Air Traffic Management, 6th edition

[12] ICAO Doc 10039 (Manual on System Wide Information Management (SWIM) Concept)

[13] ICAO Annex 10 (Aeronautical Telecommunications)

[14] ICAO Annex 11, Air Traffic Services, 15th edition

[15] ICAO PANS ATM (Procedures for Air Navigation Services Air Traffic Management)

[16] Manual on System Wide Information Management (SWIM) Concept, Doc10039 AN/511

Appendix A. Requirements Mapping

Requirements	Implementation Status	Explanation
Session shall maintain structured attributes	Completed	Attributes like aircraft ID, session ID, and session status. Context fields will be added by DLCM.
Session data shall be dynamically updated	Partially Implemented	Session updates (e.g., reassignment and reinitiation) are dynamically handled. DLCM is required to handle context association/updates.
Seamless session continuity across ATSUs	Partially Implemented	Handover mechanisms maintain continuity across ATSUs through various mobility capabilities. End-to-end continuity depends on DLCM integration.
Session replication and monitoring	Partially Implemented	Monitoring is implemented (watchdog/callbacks/presence). State and session replication requires DLCM.
Cross-domain session management	Partially Implemented	Cross-domain concepts are supported through realm-aware addressing. DLCM is required for cross-realm coordination/policy.
Secure connection establishment	Completed	TLS 1.3 for transport security.
Real-time connection health monitoring	Completed	Heartbeat and ping messages are referenced, but health metric evaluations and reaction may need improvement.
Redundant multilink with automatic failover	Completed	Multilink failover logic is included. Metric-driven policy for automated switching might be added.
Terminal mobility	Completed	Terminal IDs are tracked where session agents enable reattachment to new nodes on physical relocation.
User mobility	Partially Implemented	User context is preserved and reassigned to new terminals in DIX-based session handling. DLCM is required.
Session mobility	Partially Implemented	Core part is in place. DLCM is required.
Service mobility	Partially Implemented	Service addresses are updated dynamically using session control logic. DLCM is required.
Vertical/horizontal/intra/inter handovers	Partially Implemented	DLCM is required.
Context preservation during mobility	Partially Implemented	Context field preservation is implemented using DIX Context Identifiers and replication. DLCM is required for persistence/synchronisation.
Simultaneous multilink usage	Completed	Multiple communication links and DIX routing logic.
Dynamic link selection based on metrics	Partially Implemented	Dynamic scoring and switch logic should be improved.
Link aggregation for throughput	Partially Implemented	CTP multi-streaming/multi-homing. DLCM is required.
Compact binary message format	Completed	Message structure and format.
Transport protocol independence	Completed	Abstracts the transport layer, with DIX format consistent across protocols.
End-to-end encryption	Partially Implemented	TLS is used and existing encryption protocols can be used.
Integrity, anti-replay, tamper detection	Partially Implemented	Transport-level integrity/anti-replay via TLS.
Support PMIPv6 and GB-LISP	Partially Implemented	PMIPv6 and GB-LISP support requires adapters to be used.

Error detection, reporting, recovery	Completed	DIX retransmission flags and response codes handle some errors. Error detection, reporting, and recovery mechanisms.
Automatic retries and failover	Completed	Acknowledgments and retries along with failover strategies are considered.