Moving Picture, Audio and Data Coding
by Artificial Intelligence
www.mpai.community

MPAI Technical Specification

Artificial Intelligence Framework
MPAI-AIF

V2.1

WARNING

Use of the technologies described in this Technical Specification may infringe patents, copyrights or intellectual property rights of MPAI Members or non-members.

MPAI and its Members accept no responsibility whatsoever for damages or liability, direct or consequential, which may result from the use of this Technical Specification.

Readers are invited to review **Error! Reference source not found.**Notices and Disclaimers.

Technical Specification
Artificial Intelligence Framework (MPAI-AIF) V2.1

1 Foreword

The international, unaffiliated, non-profit ***Moving Picture, Audio, and Data Coding by Artificial Intelligence (MPAI)*** organisation was established in September 2020 in the context of:

1. **Increasing** use of Artificial Intelligence (AI) technologies applied to a broad range of domains affecting millions of people
2. **Marginal** reliance on standards in the development of those AI applications

3. **Unprecedented** impact exerted by standards on the digital media industry affecting billions of people

believing that AI-based data coding standards will have a similar positive impact on the Information and Communication Technology industry.

The design principles of the MPAI organisation as established by the MPAI Statutes are the development of AI-based Data Coding standards in pursuit of the following policies:

1. Publish upfront clear Intellectual Property Rights licensing frameworks.
2. Adhere to a rigorous standard development process.
3. Be friendly to the AI context but, to the extent possible, remain agnostic to the technology thus allowing developers freedom in the selection of the more appropriate – AI or Data Processing – technologies for their needs.
4. Be attractive to different industries, end users, and regulators.
5. Address five standardisation areas:
    1. *Data Type*, a particular type of Data, e.g., Audio, Visual, Object, Scenes, and Descriptors with as clear semantics as possible.
    2. *Qualifier*, specialised Metadata conveying information on Sub-Types, Formats, and Attributes of a Data Type.
    3. *AI Module* (AIM), processing elements with identified functions and input/output Data Types.
    4. *AI Workflow* (AIW), MPAI-specified configurations of AIMs with identified functions and input/output Data Types.
    5. *AI Framework* (AIF), an environment enabling dynamic configuration, initialisation, execution, and control of AIWs.
6. Provide appropriate Governance of the ecosystem created by MPAI Technical Specifications enabling users to:
    1. *Operate* Reference Software Implementations of MPAI Technical Specifications provided together with Reference Software Specifications
    2. *Test* the conformance of an implementation with a Technical Specification using the Conformance Testing Specification.
    3. *Assess* the performance of an implementation of a Technical Specification using the Performance Assessment Specification.
    4. *Obtain* conforming implementations possibly with a performance assessment report from a trusted source through the MPAI Store.

Today, the MPAI organisation operated on four solid pillars:

1. The MPAI Patent Policy specifies the MPAI standard development process and the Framework Licence development guidelines.
2. *Technical Specification: Artificial Intelligence Framework (MPAI-AIF)* V2.1 specifies an environment enabling initialisation, dynamic configuration, and control of AIWs in the standard AI Framework environment depicted in Figure 1. An AI Framework can execute AI applications called AI Workflows (AIW) typically including interconnected AI Modules (AIM). MPAI-AIF supports small- and large-scale high-performance components and promotes solutions with improved explainability.

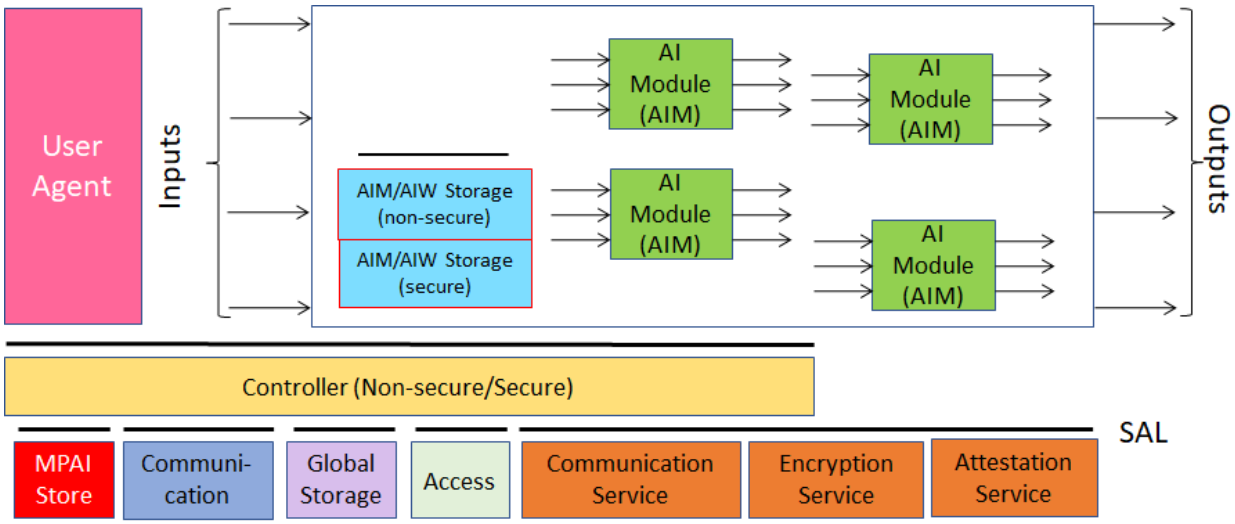

Figure 1 – The AI Framework (MPAI-AIF) V2 Reference Model

3. ***Technical Specification: Data Types, Formats, and Attributes (MPAI-TFA) V1.2*** specifies Qualifiers, a type of metadata supporting the operation of AIMs receiving data from other AIMs. Qualifiers convey information on Sub-Types (e.g., the type of colour), Formats (e.g., the type of compression and transport), and Attributes (e.g., semantic information in the Content). Although Qualifiers are human-readable, they are only intended to be used by AIMs. Therefore, Text, Speech, Audio, Visual, and other Data exchanged by AIWs and AIMs should be interpreted as being composed of Content (Text, Speech, Audio, and Visual as appropriate) and associated Qualifiers. Therefore a Text Object is composed of Text Data and Text Qualifier. The specification of most MPAI Data Types reflects this point.
4. ***Technical Specification: Governance of the MPAI Ecosystem (MPAI-GME) V1.1*** defines the following elements:
    1. Standards, i.e., the ensemble of Technical Specifications, Reference Software, Conformance Testing, and Performance Assessment.
    2. Developers of MPAI-specified AIMs and Integrators of MPAI-specified AIWS (Implementers).
    3. MPAI Store in charge of making AIMs and AIWs submitted by Implementers available to Integrators and End Users.
    4. Performance Assessors, independent entities assessing the performance of implementations in terms of Reliability, Replicability, Robustness, and Fairness.
    5. End Users.

The interaction between and among actors of the MPAI Ecosystem are depicted in Figure 2.

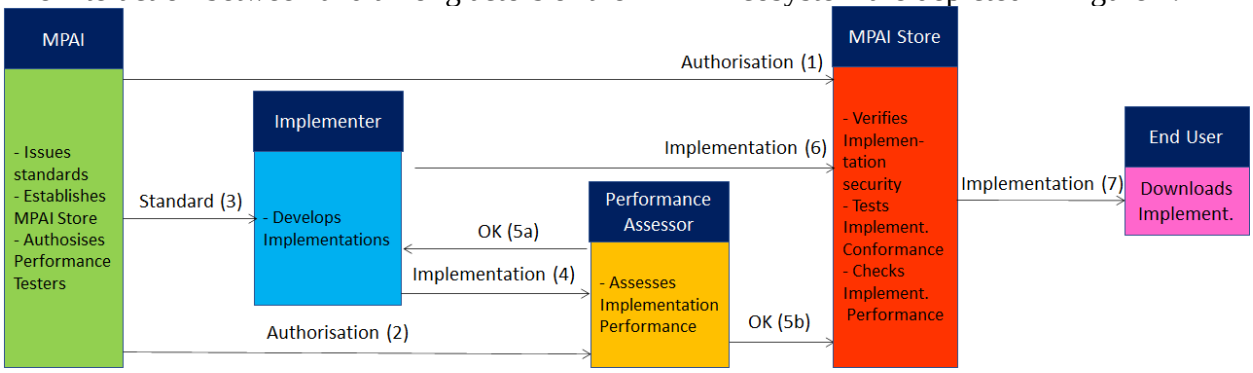

Figure 2 – The MPAI Ecosystem

2 Introduction (Informative)

Technical Specification: Artificial Intelligence Framework (MPAI-AIF) V2.1 – in the following also called MPAI-AIF V2.1 or simply MPAI-AIF – provides a standard environment where AI Workflows (AIW) composed of AI Modules are initialised, dynamically configured, executed and controlled. AIWs can be standardised by MPAI, i.e., they perform standardised functions, expose standard interfaces, and execute explicit computing workflows, or can proprietary provided they expose the interfaces specified by MPAI-AIF. Developers can compete in providing AIF components – AWs and AIM – that have standard functions and interfaces that may have improved performance compared to other implementations. AIMs can execute data processing or Artificial Intelligence algorithms and can be implemented in hardware, software, or hybrid hardware/software.

The AI Framework specified by MPAI-AIF offers the following basic functionalities:

- **Independence** of Operating System.
- **Modularity** component-based architecture with specified interfaces.
- **Encapsulation** of Components to abstract Interfaces from the development environment.
- **Access** to validated Components in the MPAI Store.
- **Implementation** of Component as:
  - **Software only**, from MCUs to HPC.
  - **Hardware only**.
  - **Hybrid** hardware-software.
- **Execution** in local and distributed Zero-Trust architectures.
- **Interaction** with other Implementations operating in proximity.
- Support for **Machine Learning** functionalities.

Since MPAI-AIF V2.0, the Specification provides access to the following Trusted Services:

- A selected range of **cyphering algorithms**.
- A basic **attestation function**.
- **Secure storage** (RAM, internal/external flash, or internal/external/remote disk).
- Certificate-based **secure communication**.
- The AIF can **execute only one AIW containing only one AIM** that following features:
  - One AIM that may be a **Composite AIM**.
  - The **AIMs** of the Composite AIM **cannot access the Security API**.
- ´The AIF Trusted Services may **rely on hardware and OS security features** already existing in the hardware and software of the environment in which the AIF is implemented.

Various actors – developers, integrators, and end users – benefit from the creation-composition-execution-update of AIM-based workflows interconnecting multi-vendor AIMs trained to specific tasks, operating in the *standard* AI framework and exchanging data in *standard* formats:

- **Technology providers** can offer standard-conforming AI technologies to an open market
- **Application developers** can find the technologies they need on the market.
- **Innovation** is fueled by demand for novel/ more performing AI components
- **Consumers** have a wider choice of better AI applications from a competitive market
- **Society** can lift the veil of opacity from large, monolithic AI-based applications.

AIW and its AIMs may have 3 interoperability levels:

1. *Level 1* – Proprietary and satisfying the MPAI-AIF Standard.
2. *Level 2* – Specified by an MPAI Application Standard.
3. *Level 3* – Specified by an MPAI Application Standard and certified by a Performance Assessor.

MPAI offers Users access to the promised benefits of AI with a guarantee of increased transparency, trust and reliability as the Interoperability Level of an Implementation moves from 1 to 3.

The chapters and the annexes of this Technical Specification are Normative unless they are labelled as Informative. Terms beginning with a capital letter are defined in *Table 1* if specific of this MPAI-AIF Technical Specification, or in *Table 2* is used across MPAI Standards.

3 Scope

***Technical Specification: AI Framework* (MPAI-AIF) V2.1** – in the following also called MPAI-AIF V2 or simply MPAI-AIF – specifies the architecture, interfaces, protocols, and Application Programming Interfaces (API) of an AI Framework specially designed for execution of AI-based implementations, but also suitable for mixed AI and traditional data processing workflows.

The current version of the Technical Specification: AI Framework (MPAI-AIF) V2 has been developed by the MPAI AI Framework Development Committee (AIF-DC). Future Versions may revise and/or extend the Scope of the Technical Specification.

4 Definitions

Terms beginning with a capital letter have the meaning defined in Table 1. Terms beginning with a small letter have the meaning commonly defined for the context in which they are used. For instance, Table 1 defines *Object* and *Scene* but does not define *object* and *scene*.
A dash "-" preceding a Term in Table 1 indicates the following readings according to the font:

1. Normal font: the Term in the table without a dash and preceding the one with a dash should be read before that Term. For example, "Avatar" and "- Model" will yield "Avatar Model."
2. *Italic* font: the Term in the table without a dash and preceding the one with a dash should be read after that Term. For example, "Avatar" and "- Portable" will yield "Portable Avatar."

Table 1 – General MPAI-AIF terms

Term	Definition
Access	Static or slowly changing data that are required by an application such as domain knowledge data, data models, etc.
AI Framework (AIF)	The environment where AIWs are executed.
AI Module (AIM)	A processing element receiving AIM-specific Inputs and producing AIM-specific Outputs according to according to its Function. An AIM may be an aggregation of AIMs. AIMs operate in the Trusted Zone.
AI Workflow (AIW)	A structured aggregation of AIMs implementing a Use Case receiving AIM-specific inputs and producing AIM-specific outputs according to its Function. AIWs operate in the Trusted Zone.
AIF Metadata	The data set describing the capabilities of an AIF set by the AIF Implementer.
AIM Metadata	The data set describing the capabilities of an AIM set by the AIM Implementer.
AIM Storage	A Component to store data of individual AIMs. An AIM may only access its own data. The AIM Storage is part of the Trusted Zone.
AIW Metadata	The data set describing the capabilities of an AIW set by the AIW Implementer.
Channel	A physical or logical connection between an output Port of an AIM and an input Port of an AIM. The term "connection" is also used as synonymous. Channels are part of the Trusted Zone.

Communication	The infrastructure that implements message passing between AIMs. Communication operates in the Trusted Zone.
Component	One of the 9 AIF elements: Access, AI Module, AI Workflow, Communication, Controller, AIM Storage, Shared Storage, Store, and User Agent.
Composite AIM	An AIM aggregating more than one AIM.
Controller	A Component that manages and controls the AIMs in the AIWs, so that they execute in the correct order and at the time when they are needed. The Controller operates in the Trusted Zone.
Data Type	An instance of the Data Types defined by 6.1.1.
Device	A hardware and/or software entity running at least one instance of an AIF.
Event	An occurrence acted on by an Implementation.
External Port	An input or output Port simulating communication with an external Controller.
Group Element	An AIF in a in a proximity-based scenario.
Knowledge Base	Structured and/or unstructured information made accessible to AIMs via MPAI-specified interfaces.
Message	A sequence of Records.
MPAI Ontology	A dynamic collection of terms with a defined semantics managed by MPAI.
MPAI Server	A remote machine executing one or more AIMs.
Remote Port	A Port number associated with a specific remote AIM.
Store	The repository of Implementations.
Port	A physical or logical communication interface of an AIM.
Record	Data with a specified Format.
Resource policy	The set of conditions under which specific actions may be applied.
Security Abstraction Layer	(SAL) The set of Trusted Services that provide security functionalities to AIF.
Shared Storage	A Component to store data shared among AIMs. The Shared Storage is part of the Trusted Zone.
Status	The set of parameters characterising a Component.
Structure	A composition of Records
Time Base	The protocol specifying how Components can access timing information. The Time Base is part of the Trusted Zone.
Topology	The set of Channels connecting AIMs in an AIW.
Trusted Zone	An environment that contains only trusted objects, i.e., object that do not require further authentication.
User Agent	The Component interfacing the user with an AIF through the Controller
Zero Trust	A cybersecurity model primarily focused on data and service protection that assumes no implicit trust [28].
Security Abstraction Layer	A layer acting as a bridge between the AIMs and the Control on one side, and the security functions.

Table 2 – MPAI-wide definitions

The Terms used in this standard whose first letter is capital and are not already included in *Table 1* are defined in *Table 2*.

Note: To concentrate in one place all the Terms that are composed of a common name followed by other words (e.g., the word Data followed by one of the words Format, Type, or Semantics), the definition given to a Terms preceded by a dash "-" applies to a Term composed by that Term without the dash preceded by the Term that precedes it in the column without a dash.

Table 2 – MPAI-wide Terms

Term	**Definition**
Access	Static or slowly changing data that are required by an application such as domain knowledge data, data models, etc.
AI Framework (AIF)	The environment where AIWs are executed.
AI Model (AIM)	A data processing element receiving AIM-specific Inputs and producing AIM-specific Outputs according to according to its Function. An AIM may be an aggregation of AIMs.
AI Workflow (AIW)	A structured aggregation of AIMs implementing a Use Case receiving AIW-specific inputs and producing AIW-specific outputs according to the AIW Function.
Application Standard	An MPAI Standard designed to enable a particular application domain.
Channel	A connection between an output port of an AIM and an input port of an AIM. The term "connection" is also used as synonymous.
Communication	The infrastructure that implements message passing between AIMs.
Component	One of the 7 AIF elements: Access, Communication, Controller, Internal Storage, Global Storage, Store, and User Agent
Composite AIM	An AIM aggregating more than one AIM.
Component	One of the 7 AIF elements: Access, Communication, Controller, Internal Storage, Global Storage, Store, and User Agent
Conformance	The attribute of an Implementation of being a correct technical Implem-entation of a Technical Specification.
– Testing	The normative document specifying the Means to Test the Conformance of an Implementation.
– Testing Means	Procedures, tools, data sets and/or data set characteristics to Test the Conformance of an Implementation.
Connection	A channel connecting an output port of an AIM and an input port of an AIM.
Controller	A Component that manages and controls the AIMs in the AIF, so that they execute in the correct order and at the time when they are needed
Data	Information in digital form.
– Format	The standard digital representation of Data.
– Type	An instance of Data with a specific Data Format.
– Semantics	The meaning of Data.
Descriptor	Coded representation of a text, audio, speech, or visual feature.
Digital Representation	Data corresponding to and representing a physical entity.
Ecosystem	The ensemble of actors making it possible for a User to execute an application composed of an AIF, one or more AIWs, each with one or more AIMs potentially sourced from independent implementers.
Explainability	The ability to trace the output of an Implementation back to the inputs that have produced it.
Fairness	The attribute of an Implementation whose extent of applicability can be assessed by making the training set and/or network open to testing for bias and unanticipated results.
Function	The operations effected by an AIW or an AIM on input data.
Global Storage	A Component to store data shared by AIMs.
AIM/AIW Storage	A Component to store data of the individual AIMs.
Identifier	A name that uniquely identifies an Implementation.
Implementation	1. An embodiment of the MPAI-AIF Technical Specification, or

	2. An AIW or AIM of a particular Level (1-2-3) conforming with a Use Case of an MPAI Application Standard.
Implementer	A legal entity implementing MPAI Technical Specifications.
ImplementerID (IID)	A unique name assigned by the ImplementerID Registration Authority to an Implementer.
ImplementerID Registration Authority (IIDRA)	The entity appointed by MPAI to assign ImplementerID's to Implementers.
Instance ID	Instance of a class of Objects and the Group of Objects the Instance belongs to.
Interoperability	The ability to functionally replace an AIM with another AIW having the same Interoperability Level
– Level	The attribute of an AIW and its AIMs to be executable in an AIF Implementation and to: 1. Be proprietary (Level 1) 2. Pass the Conformance Testing (Level 2) of an Application Standard 3. Pass the Performance Testing (Level 3) of an Application Standard.
Knowledge Base	Structured and/or unstructured information made accessible to AIMs via MPAI-specified interfaces
Message	A sequence of Records transported by Communication through Channels.
Normativity	The set of attributes of a technology or a set of technologies specified by the applicable parts of an MPAI standard.
Performance	The attribute of an Implementation of being Reliable, Robust, Fair and Replicable.
– Assessment	The normative document specifying the Means to Assess the Grade of Performance of an Implementation.
– Assessment Means	Procedures, tools, data sets and/or data set characteristics to Assess the Performance of an Implementation.
– Assessor	An entity Assessing the Performance of an Implementation.
Profile	A particular subset of the technologies used in MPAI-AIF or an AIW of an Application Standard and, where applicable, the classes, other subsets, options and parameters relevant to that subset.
Record	A data structure with a specified structure
Reference Model	The AIMs and theirs Connections in an AIW.
Reference Software	A technically correct software implementation of a Technical Specification containing source code, or source and compiled code.
Reliability	The attribute of an Implementation that performs as specified by the Application Standard, profile, and version the Implementation refers to, e.g., within the application scope, stated limitations, and for the period of time specified by the Implementer.
Replicability	The attribute of an Implementation whose Performance, as Assessed by a Performance Assessor, can be replicated, within an agreed level, by another Performance Assessor.
Robustness	The attribute of an Implementation that copes with data outside of the stated application scope with an estimated degree of confidence.
Scope	The domain of applicability of an MPAI Application Standard

Service Provider	An entrepreneur who offers an Implementation as a service (e.g., a recommendation service) to Users.
Standard	A set of Technical Specification, Reference Software, Conformance Testing, Performance Assessment, and Technical Report of an MPAI application Standard.
Technical Specification	(Framework) the normative specification of the AIF. (Application) the normative specification of the set of AIWs belonging to an application domain along with the AIMs required to Implement the AIWs that includes: 1. The formats of the Input/Output data of the AIWs implementing the AIWs. 2. The Connections of the AIMs of the AIW. 3. The formats of the Input/Output data of the AIMs belonging to the AIW.
Testing Laboratory	A laboratory accredited to Assess the Grade of Performance of Implementations.
Time Base	The protocol specifying how Components can access timing information
Topology	The set of AIM Connections of an AIW.
Use Case	A particular instance of the Application domain target of an Application Standard.
User	A user of an Implementation.
User Agent	The Component interfacing the user with an AIF through the Controller
Version	A revision or extension of a Standard or of one of its elements.
Zero Trust	A cybersecurity model primarily focused on data and service protection that assumes no implicit trust.

5 References

5.1 Normative References

MPAI-AIF normatively references the following documents:

1. MPAI; The MPAI Statutes; https://mpai.community/statutes/
2. MPAI; The MPAI Patent Policy; https://mpai.community/about/the-mpai-patent-policy/.
3. MPAI; Technical Specification: Governance of the MPAI Ecosystem; https://mpai.community/standards/mpai-gme/
4. GIT protocol, https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols.
5. ZIP format, https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.
6. Date and Time in the Internet: Timestamps; IETF RFC 3339; July 2002.
7. Uniform Resource Identifiers (URI): Generic Syntax, IETF RFC 2396, August 1998.
8. The JavaScript Object Notation (JSON) Data Interchange Format; https://datatracker.ietf.org/doc/html/rfc8259; IETF rfc8259; December 2017
9. JSON Schema; https://json-schema.org/.
10. BNF Notation for syntax; https://www.w3.org/Notation.html
11. MPAI; The MPAI Ontology; https://mpai.community/standards/mpai-aif/mpai-ontology/
12. Framework Licence of the Artificial Intelligence Framework Technical Specification (MPAI-AIF); https://mpai.community/standards/mpai-aif/framework-licence/
13. Bormann, C. and P. Hoffman, Concise Binary Object Representation (CBOR), December 2020. https://rfc-editor.org/info/std94

14. Schaad, J., CBOR Object Signing and Encryption (COSE): Structures and Process, August 2022. https://rfc-editor.org/info/std96
15. IETF Entity Attestation Token (EAT), Draft. https://datatracker.ietf.org/doc/draft-ietf-rats-eat
16. IEEE, 1619-2018 — IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices, January 2019. https://ieeexplore.ieee.org/servlet/opac?punumber=8637986
17. IETF, The MD5 Message-Digest Algorithm, April 1992. https://tools.ietf.org/html/rfc1321.html
18. [RFC6979] IETF, Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), August 2013. https://tools.ietf.org/html/rfc6979.html
19. [RFC7539] IETF, ChaCha20 and Poly1305 for IETF Protocols, May 2015. https://tools.ietf.org/html/rfc7539.html
20. [RFC7919] IETF, Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS), August 2016. https://tools.ietf.org/html/rfc7919.html
21. [RFC8017] IETF, PKCS #1: RSA Cryptography Specifications Version 2.2, November 2016. https://tools.ietf.org/html/rfc8017.html
22. [RFC8032] IRTF, Edwards-Curve Digital Signature Algorithm (EdDSA), January 2017. https://tools.ietf.org/html/rfc8032.html
23. Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography, May 2009. https://www.secg.org/sec1-v2.pdf
24. NIST, *FIPS Publication 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, August 2015.https://doi.org/10.6028/NIST.FIPS.202
25. NIST, NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, December 2001. https://doi.org/10.6028/NIST.SP.800-38A
26. NIST, NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, November 2007. https://doi.org/10.6028/NIST.SP.800-38D

5.2 Informative References

27. Message Passing Interface (MPI), https://www.mcs.anl.gov/research/projects/mpi/
28. Rose, Scott; Borchert, Oliver; Mitchell, Stu; Connelly, Sean; "Zero Trust Architecture"; https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf
29. MPAI Technical Specification: Context-based Audio Enhancement (MPAI-CAE) V2; https://mpai.community/standards/mpai-cae/.
30. MPAI Technical Specification: Connected Autonomous Vehicle – Architecture (MPAI-CAV) V1; https://mpai.community/standards/mpai-cav/.
31. MPAI Technical Specification: Compression and Understanding of Industrial Data (MPAI-CUI) V1.1; https://mpai.community/standards/mpai-cui/.
32. MPAI Technical Specification: Multimodal Conversation (MPAI-MMC) V2; https://mpai.community/standards/mpai-mmc/.
33. MPAI Technical Specification: Neural Network Watermarking (MPAI-MMC) V1; https://mpai.community/standards/mpai-nnw/.
34. MPAI Technical Specification: Portable Avatar Format (MPAI-PAF) V1; https://mpai.community/standards/mpai-paf/.

35. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad, "Rafiki: machine learning as an analytics service system," Proceedings of the VLDB Endowment, vol. 12, no. 2, pp. 128–140, 2018.
36. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi; PRETZEL: Opening the black box of machine learning prediction serving systems; in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI18), pp. 611–626, 2018.
37. NET [ONLINE]; https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet.
38. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica; Clipper: A low-latency online prediction serving system; in NSDI, pp. 613–627, 2017.
39. Zhao, M. Talasila, G. Jacobson, C. Borcea, S. A. Aftab, and J. F. Murray; Packaging and sharing machine learning models via the acumos ai open platform; in 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 841–846, IEEE, 2018.
40. Apache Prediction I/O; https://predictionio.apache.org/.
41. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, D. Dennison; Hidden technical debt in Machine learning systems Share; on NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems – Volume 2; December 2015 Pages 2503–2511
42. Arm; "PSA Certified Crypto API 1.1," IHI 0086, issue 2,23/03/2022, https://arm-software.github.io/psa-api/crypto/1.1/
43. Arm; "PSA Certified Secure Storage API 1.0," IHI 0087, issue 2, 23/03/2023, https://arm-software.github.io/psa-api/storage/1.0/
44. Arm; "PSA Certified Attestation API 1.0," IHI 0085, issue 3, 17/10/2022, https://arm-software.github.io/psa-api/attestation/1.0/

6 Architecture

6.1 AI Framework Components

This MPAI-AIF Version adds a Secure Profile with Security functionalities on top of the Basic Profile of Version 1.1 with the following restrictions:

- There is only one AIW containing only one AIM – which may be a Composite AIM.
- The AIM implementer guarantees the security of the AIM by calling the security API.
- The AIF application developer cannot access securely the Composite AIM internals.

6.1.1 Components for Basic Functionalities

Figure 1 specifies the MPAI-AIF Components supported by MPAI-AIF Version 2.0.

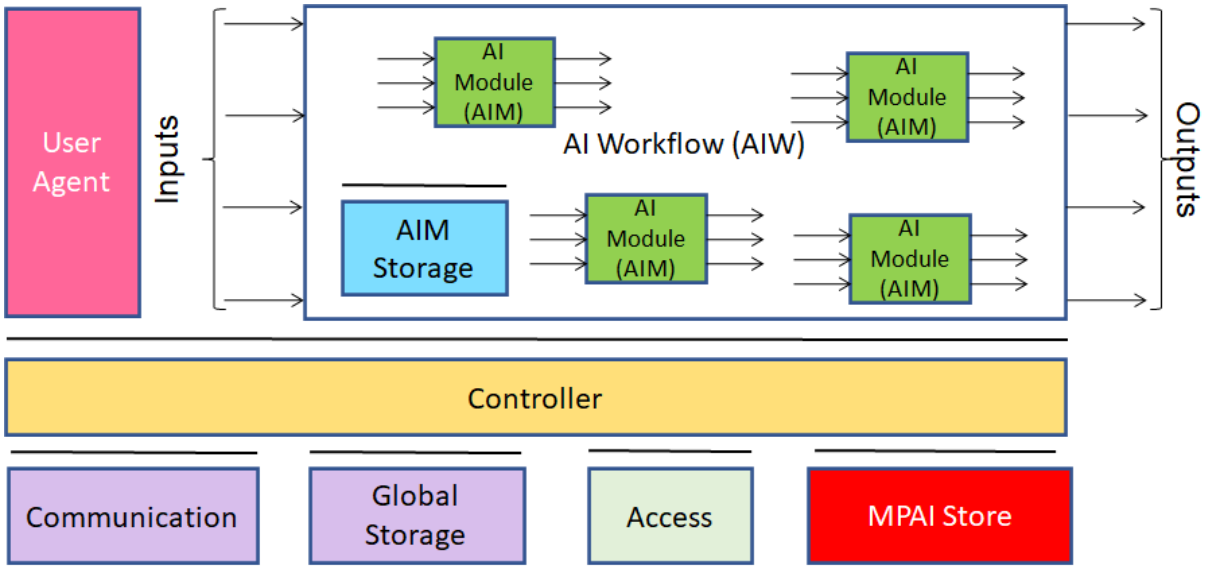

Figure 1 – The MPAI-AIF V1 Reference Model

The specific functions of the Components are:

1. **Controller**:
    - Provides basic functionalities such as scheduling, communication between AIMs and with AIF Components such as AIM Storage and Global Storage.
    - Acts as a resource manager, according to instructions given by the User through the User Agent.
    - Can interact by default to all the AIMs in a given AIF.
    - Activates/suspends/resumes/deactivates AIWs based on User's or other inputs.
    - May supports complex application scenarios by balancing load and resources.
    - Accesses the *MPAI Store APIs* to download AIWs and AIMs.
    - Exposes three APIs:
        - *AIM APIs* enable AIMs to communicate with it (register themselves, communicate and access the rest of the AIF environment). An AIW is an AIM with additional metadata. Therefore, an AIW uses the same AIM API.
        - *User APIs* enable User or other Controllers to perform high-level tasks (e.g., switch the Controller on and off, give inputs to the AIW through the Controller).
        - *Controller-to-Controller API* enables interactions among Controllers.
    - May run an AIW on different computing platforms and may run more than one AIW.
    - May communicate with other Controllers.
2. **Communication**: connects the AIF Components via Events or Channels connecting an output Port of an AIM with an input Port of another AIM. Communication has the following characteristics:
    - The Communication Component is turned on jointly with the Controller.
    - The Communication Component needs not be persistent.
    - Channels are unicast and may be physical or logical.
    - Messages are transmitted via Channels. They are composed of sequences of Records and may be of two types:

        - High-Priority Messages expressed as up to 16-bit integers.
        - Normal-Priority Messages expressed as MPAI-AIF defined types (6.1.1).
            - Messages may be communicated through Channels or Events.
4. **AI Module** (AIM): a data processing element with a specified Function receiving AIM-specific inputs and producing AIM-specific outputs having the following characteristics:
    - Communicates with other Components through Ports or Events.
    - Includes at least one input Port and one output Port.
    - May incorporate other AIMs.
    - May be hot-pluggable, and dynamically register and disconnect itself on the fly.
    - May be executed:
        - Locally, e.g., it encapsulates hardware physically accessible to the Controller.
        - On different computing platforms, e.g., in the cloud or on groups of drones, and encapsulates communication with a remote Controller.
5. **AI Workflow** (AIW): an organised aggregation of AIMs receiving AIM-specific inputs and producing AIM-specific outputs according to its Function implementing a Use Case that is either proprietary or specified by an MPAI Application Standard.
6. **Global Storage**: stores data shared by AIMs.
7. **AIM Storage**: stores data of individual AIMs.
8. **User Agent**: interfaces the User with an AIF through the Controller.
9. **Access**: provides access to static or slowly changing data that is required by AIMs such as domain knowledge data, data models, etc.
10. **MPAI Store**: stores Implementations for users to download by secure protocols.

Note: When different Controllers running on separate computing platforms (Group Elements) interact with one another, they cooperate by requesting one or more Controllers in range to open Remote Ports. The Controllers on which the Remote Ports are opened can then react to information sent by other Controllers in range through the Remote Ports and implement a collective behaviour of choice. For instance: there is a main Controller and the other Controllers in the Group react to the information it sends; or there is no main Controller and all Controllers in the Group behave according to a collective logic specified in the Controllers.

6.1.2 Components for Security Functionalities

The AIF Components have the following features:

1. **The AIW**
    - The AIMs in the AIW trust each other and communicate without special security concerns.
    - Communication among AIMs in the Composite AIM is non-secure.
2. **The Controller**
    - Communicates securely with the MPAI-Store and the User Agent (Authentication, Attestation, and Encryption).
    - Accesses Communication, Global Storage, Access and MPAI Store via Trusted Services API.
    - Is split in two parts:
        - Secure Controller accesses Secure Communication and Secure Storage.
        - Non-Secure Controller can access the non-secure parts of the AIF.
    - Interfaces with the User Agent in the area where non-secure code is executed.
    - Interface with the Composite AIM in the area where secure code is executed,
3. **AIM/AIW Storage**
    - Secure Storage functionality is provided through key exchange.

- Non-secure functionality is provided without reference to secure API calls.

4. **The AIW/AIMs** call the Secure Abstraction Layer via API.
5. The **AIMs of a Composite AIM** shall run on the same computing platform.

Figure 3 specifies the MPAI-AIF Components operating in the secure environment created by the Secure Abstraction Layer.

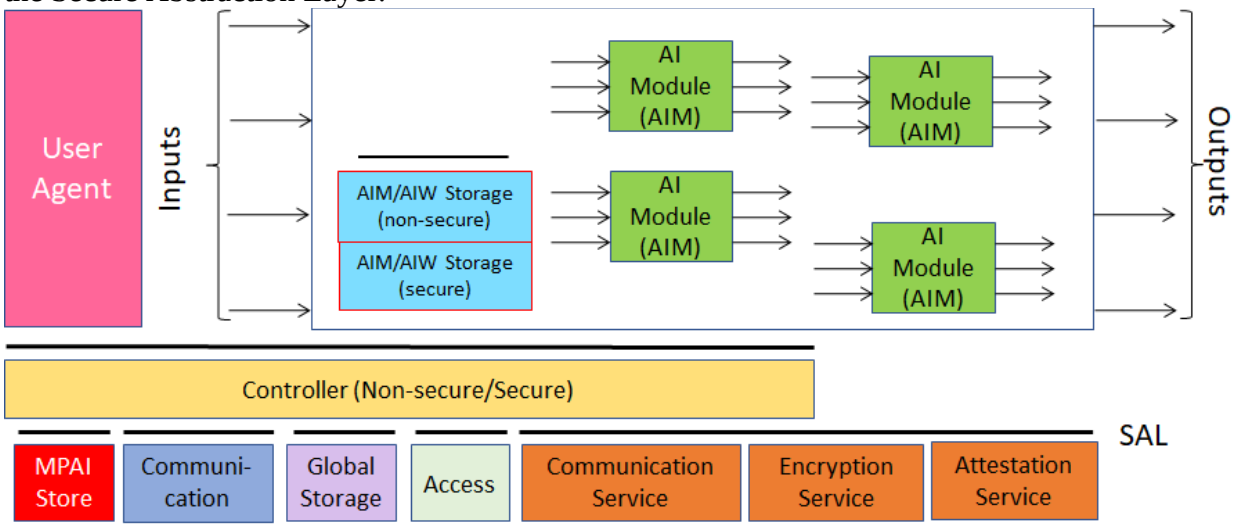

Figure 2 – The MPAI-AIF V2 Reference Model

6.2 AI Framework Implementations

MPAI-AIF enables a wide variety of Implementations:

1. AIF Implementations can be tailored to different execution environments, e.g., High-Performance Computing systems or resource-constrained computing boards. For instance, the Controller might be a process on a HPC system or a library function on a computing board.
2. There is always a Controller even if the AIF is a lightweight Implementation.
3. The API may have different MPAI-defined Profiles to allow for Implementations:
   - To run on different computing platforms and different programming languages.
   - To be based on different hardware and resources available.
4. AIMs may be Implemented in hardware, software and mixed-hardware and software.
5. Interoperability between AIMs is ensured by the way communication between AIMs is defined, irrespective of whether they are implemented in hardware or software.
6. Use of Ports and Channels ensures that compatible AIM Ports may be connected irrespective of the AIM implementation technology.
7. Message generation and Event management is implementation independent.

6.3 AIMs

6.3.1 Implementation types

AIMs can be implemented in either hardware or software keeping the same interfaces independent of the implementation technology. However, the nature of the AIM might impose constraints on the specific values of certain API parameters and different Profiles may impose different constraints. For instance, Events (easy to accommodate in software but less so in hardware); and persistent Channels (easy to make in hardware, less so in software).
While software-software and hardware-hardware connections are homogeneous, a hybrid hardware-software scenario is inherently heterogeneous and requires the specification of

additional communication protocols, which are used to wrap the hardware part and connect it to software. A list of such protocols is provided by the MPAI Ontology [11].
Examples of supported architectures are:

- *CPU-based devices* running an operating system.
- *Memory-mapped devices* (FPGAs, GPUs, TPUs) which are presented as accelerators.
- *Cloud-based frameworks*.
- *Naked hardware devices* (i.e., IP in FPGAs) that communicate through hardware Ports.
- *Encapsulated blocks of a hardware design* (i.e., IP in FPGAs) that communicate through a memory-mapped bus. In this case, the Metadata associated with the AIM (see 6.3) shall also specify the low-level communication protocol used by the Ports.

6.3 Combination

MPAI-AIF supports the following ways of combining AIMs:

- *Software AIMs* connected to other software AIMs resulting in a software AIM.
- *Non-encapsulated hardware blocks* connected to other non-encapsulated hardware blocks, resulting in a larger, non-encapsulated hardware AIM.
- *Encapsulated hardware blocks* connected to either other encapsulated hardware blocks or other software blocks, resulting in a larger software AIM.

Connection between a non-encapsulated hardware AIM and a software AIM is not supported as in such a case direct communication between the AIMs cannot be defined in any meaningful way.

6.3.1 Hardware-software compatibility

To achieve communication among AIMs irrespective of their implementation technology, the requirements considered in the following two cases should be satisfied:

1. *Hardware AIM to Hardware AIM*: Each named type in a Structure is transmitted as a separate channel. Vector types are implemented as two channels, one transmitting the size and the second transmitting the data.
2. *All other combinations*: Fill out a Structure by recursively traversing the definition (breadth-first). Sub-fields are laid down according to their type, in little-endian order.

6.3.2 Actual implementations

6.3.3 Hardware

Metadata ensures that hardware blocks can be directly connected to other hardware/software blocks, provided the specification platforms for the two blocks have compatible interfaces, i.e., they have compatible Ports and Channels.

6.3.4 Software

Software Implementations shall ensure that Communication among different constituent AIMs, and with other AIMs outside the block, is performed correctly.
In addition, AIM software Implementations shall contain a number of well-defined steps so as to ensure that the Controller is correctly initialised and remains in a consistent internal state, i.e.:

1. **Code registering the different AIMs** used by the AIW. The registration operation specifies where the AIMs will be executed, either locally or remotely. The AIM Implementations are archives downloaded from the Store containing source code, binary code and hardware designs executed on a local machine/HPC cluster/MPC machine or a remote machine.
2. **Code starting/stopping** the AIMs.
3. **Code registering the input/output Ports** for the AIM.

4. **Code instantiating unicast channels** between AIM Ports belonging to AIMs used by the AIW, and connections from/to the AIM being defined to/from remote AIMs.
5. **Registering Ports** and connecting them may result in a number of steps performed by the Controller – some suitable data structure (including, for instance, data buffers) will be allocated for each Port or Channel, in order to support the functions specified by the Controller API called by the AIM (8.3).
6. **Explicitly write/read data** to/from, any of the existing Ports.
7. In general, arbitrary functionality can be added to a software AIM. For instance, depending on the AIM Function, one would typically link libraries that allow a GPU or FPGA to be managed through Direct Memory Access (DMA), or link and use high-level libraries (e.g., TensorFlow) that implement AI-related functionality.
8. The API implementation depends on the architecture the Implementation is designed for.

7 Metadata

Metadata specifies static properties pertaining to the interaction between:

1. A Controller and its hosting hardware.
2. An AIW and the Controller hosting it.
3. An AIW and its composing AIMs.

Metadata specified in the following Sections is represented in JSON Schema.

7.1 Communication channels and their data types

This Section specifies how Metadata pertaining to a communication Channel is defined.

7.1.1 Type system

The data interchange happening through buffers involves the exchange of structured data. Message data types exchanged through Ports and communication Channels are defined by the following Backus–Naur Form (BNF) specification [10]. Words in bold typeface are keywords; capitalised words such as NAME are tokens.

fifo_type :=
| /* The empty type */
| base_type NAME
recursive_type :=
| recursive_base_type NAME
base_type :=| toplevel_base_type| recursive_base_type| **(** base_type **)**toplevel_base_type :=
| array_type| toplevel_struct_type| toplevel_variant_type
array_type :=
| recursive_base_type **[]**
toplevel_struct_type :=
| **{** one_or_more_fifo_types_struct **}**
one_or_more_fifo_types_struct :=
| fifo_type
| fifo_type **;** one_or_more_fifo_types_struct
toplevel_variant_type :=
| **{** one_or_more_fifo_types_variant **}**
one_or_more_fifo_types_variant :=
| fifo_type **|** fifo_type
| fifo_type **|** one_or_more_fifo_types_variant
recursive_base_type :=

| signed_type
| unsigned_type
| float_type
| struct_type
| variant_type
signed_type :=
| **int8**
| **int16**
| **int32**
| **int64**
unsigned_type :=
| **uint8** | **byte**
| **uint16**
| **uint32**
| **uint64**
float_type :=
| **float32**
| **float64**
struct_type :=
| **{** one_or_more_recursive_types_struct **}**
one_or_more_recursive_types_struct :=
| recursive_type
| recursive_type **;** one_or_more_recursive_types_struct
variant_type :=
| **{** one_or_more_recursive_types_variant **}**
one_or_more_recursive_types_variant :=
| recursive_type **|** recursive_type
| recursive_type **|** one_or_more_recursive_types_variant

Valid types for FIFOs are those defined by the production fifo_type.

Although this syntax allows to specify types having a fixed length, the general record type written to, or read from, the Port will not have a fixed length. If an AIM implemented in hardware receives data from an AIM implemented in software the data format should be harmonised with the limitations of the hardware AIM.

7.1.2 Mapping the type to buffer contents

The Type definition allows to derive an automated way of filling and transmitting buffers both for hardware and software implementations. Data structures are turned into low-level memory buffers, filled out by recursively traversing the definition (breadth-first). Sub-fields are laid down according to their type, in little-endian order.

For instance, a definition for transmitting a video frame through a FIFO might be:

{int32 frameNumber; int16 x; int16 y; byte[] frame} frame_t

and the corresponding memory layout would be:

[32 bits: frameNumber | 16 bits: x | 16 bits: y | 32 bits: size(frame) | 8*size(frame) bits: frame].

API functions are provided to parse the content of raw memory buffers in a platform- and implementation-independent fashion (see Subsection 8.3.7).

7.2 AIF Metadata

AIF Metadata is specified in terms of JSON Schema [9] definition
at http://schemas.mpai.community/AIF/V2.0/AIF-metadata.schema.json

7.3 AIW/AIM Metadata

AIM Metadata specifies static, abstract properties pertaining to one or more AIM implementations, and how the AIM will interact with the Controller.
AIW/AIM Metadata is specified in terms of JSON Schema [9] definition
at http://schemas.mpai.community/AIF/V2.0/AIW-AIM-metadata.schema.json

8 API Conventions

The API is written in a C-like fashion. However, the specification should be meant as a definition for a general programming language.

Note that namespaces for modules, ports and communication channels (strings belonging to which are indicated in the next sections with names such as *module_name*, *port_name*, and *channel_name*, respectively) are all independent.

8.1 API types

We assume that the implementation defines several types, as follows:

message_t	the type of messages being passed through communication ports and channels
parser_t	the type of parsed message datatypes (a.k.a. “the high-level protocol”)
error_t	the type of return code defined in 7.2.2.

The actual types are opaque, and their exact definition is left to the Implementer. The only meaningful way to operate on library types with defined results is by using library functions.

On the other hand, the type of AIM Implementations, module_t, is always defined as:

```
typedef error_t *(module_t)()
```

across all implementations, in order to ensure cross-compatibility.
Types such as void, size_t, char, int, float are regular C types.

8.2 Return codes

Valid return codes:

Code	**Numeric value**
MPAI_AIM_ALIVE	1
MPAI_AIM_DEAD	2
MPAI_AIF_OK	0

Valid error codes:

Code	Semantic value
MPAI_ERROR	A generic error code
MPAI_ERROR_MEM_ALLOC	Memory allocation error
MPAI_ERROR_MODULE_NOT_FOUND	The operation requested of a module cannot be executed since the module has not been found
MPAI_ERROR_INIT	The AIW cannot be initialied
MPAI_ERROR_TERM	The AIW cannot be properly terminated
MPAI_ERROR_MODULE_CREATION_FAILED	A new AIM cannot be created
MPAI_ERROR_PORT_CREATION_FAILED	A new AIM Port cannot be created
MPAI_ERROR_CHANNEL_CREATION_FAILED	A new Channel between AIMs could not be created.
MPAI_ERROR_WRITE	A generic message writing error
MPAI_ERROR_TOO_MANY_PENDING_MESSAGES	A message writing operation failed because there are too many pending messages waiting to be delivered
MPAI_ERROR_PORT_NOT_FOUND	One or both ports of a connection has (or have) been removed
MPAI_ERROR_READ	A generic message reading error
MPAI_ERROR_OP_FAILED MPAI_ERROR_EXTERNAL_CHANNEL_CREATION_FAILED	The requested operation failed A new Channel between Controllers could not be created.

8.3 High-priority Messages

Code	Numeric value
MPAI_AIM_SIGNAL_START	1
MPAI_AIM_SIGNAL_STOP	2
MPAI_AIM_SIGNAL_RESUME	3
MPAI_AIM_SIGNAL_PAUSE	4

9 Basic API

9.1 Store API called by Controller

It is assumed that all the communication between the Controller and the Store occur via https protocol. Thus, the APIs reported refer to the http secure protocol functions (i.e. GET, POST, etc). The Store supports the GIT protocol [1].

The Controller implements the functions relative to the file retrieval as described in 9.1.1.

9.1.1 Get and parse archive

Get and parse an archive from the Store.

9.1.1.1 MPAI_AIFS_GetAndParseArchive

error_t MPAI_AIFS_GetAndParseArchive(const char* *filename*)

The default file format is tar.gz. Options are tar.gz, tar.bz2, tbz, tbz2, tb2, bz2, tar, and zip. For example, specifying archive.zip would send an archive in ZIP format [5]. The archive shall include one AIW Metadata file and one or more binary files. The parsing of JSON Metadata and the creation of the corresponding data structure is left to the Implementer.

All archives downloaded from the Store shall not leave the Trusted Zone if the AIF Profile is Basic and shall not leave the Secure Storage if the AIF Profile is Secure.

9.2 Controller API called by User Agent

9.2.1 General

This section specifies functions executed by the User Agent when interacting with the Controller. In particular:

1. Initialise all the Components of the AIF.
2. Start/Stop/Suspend/Resume AIWs.
3. Manage Resource Allocation.

9.2.1.1 MPAI_AIFU_Controller_Initialize

error_t MPAI_AIFU_Controller_Initialize()

This function, called by the User Agent, switches on and initialies the Controller, in particula

9.2.1.2 MPAI_AIFU_Controller_Destroy

error_t MPAI_AIFU_Controller_Destroy()

This function, called by the User Agent, switches off the Controller, after data structures related to running AIWs have been disposed of.

9.2.2 Start/Pause/Resume/Stop Messages to other AIWs

These functions can be used by the User Agent to send messages from the Controller to AIWs. Errors encountered while transmitting/receiving these Messages are non-recoverable – i.e., they terminate the entire AIW. AIWs can communicate with other AIWs and the Controller uses this API to Start/Pause/Resume/Stop the AIWs.

9.2.2.1 MPAI_AIFU_AIW_Start

error_t MPAI_AIFU_AIW_Start(const char* *name*, int* *AIW_ID*)

This function, called by the User Agent, registers with the Controller and starts an instance of the AIW named *name*. The AIW Metadata for *name* shall have been previously parsed. The AIW ID is returned in the variable *AIW_ID*. If the operation succeeds, it has immediate effect.

9.2.2.2 MPAI_AIFU_AIW_Pause

error_t MPAI_AIFU_AIW_Pause(int *AIW_ID*)

With this function the User Agent asks the Controller to pause the AIW with ID *AIW_ID*. If the operation succeeds, it has immediate effect.

9.2.2.3 MPAI_AIFU_AIW_Resume

error_t MPAI_AIFU_AIW_Resume(int *AIW_ID*)

With this function the User Agent asks the Controller to resume the AIW with ID *AIW_ID*. If the operation succeeds, it has immediate effect.

9.2.2.4 MPAI_AIFU_AIW_Stop

error_t MPAI_AIFU_AIW_Stop(int *AIW_ID*)

This function, called by the User Agent, deregisters and stops the AIW with ID *AIW_ID* from the Controller. If the operation succeeds, it has immediate effect.

9.2.3 Inquire about state of AIWs and AIMs

9.2.3.1 MPAI_AIFU_AIM_GetStatus

error_t MPAI_AIFU_AIM_GetStatus(int *AIW_ID*, const char* *name*, int* *status*)

With this function the User Agent inquires about the current status of the AIM named *name* belonging to AIW with ID *AIW_ID*. The status is returned in *status*. Admissible values are: MPAI_AIM_ALIVE, MPAI_AIM_DEAD.

9.2.4 Management of Shared and AIM Storage for AIWs

9.2.4.1 MPAI_AIFU_SharedStorage_Init

error_t MPAI_AIFU_SharedStorage_init(int *AIW_ID*)

With this function the User Agent initialises the Shared Storage interface for the AIW with ID *AIW_ID*.

9.2.4.2 MPAI_AIFU_ AIMStorage_Init

error_t MPAI_AIFU_ AIMStorage_init(int *AIM_ID*)

With this function the User Agent initialises the AIM Storage interface for the AIW with ID *AIW_ID*.

9.2.5 Communication management

Communication takes place with Messages communicated via Events or Ports and Channels. Their actual implementation and signal type depends on the MPAI-AIF Implementation (and hence on the specific platform, operating system, and programming language the Implementation is developed for). Events are defined AIF wide while Ports, Channels and Messages are specific to the AIM and thus part of the AIM API.

9.2.5.1 MPAI_AIFU_Communication_Event

error_t MPAI_AIFU_Communication_Event(const char* *event*)

With this function the User Agent initialises the event handling for Event named *event*.

9.2.6 Resource allocation management

9.2.6.1 MPAI_AIFU_Resource_GetGlobal

error_t MPAI_AIFU_Resource_GetGlobal(const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the User Agent interrogates the resource allocation for one AIF Metadata entry.

9.2.6.2 MPAI_AIFU_Resource_SetGlobal

error_t MPAI_AIFU_Resource_SetGlobal(const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the User Agent initialises the resource allocation for one AIF Metadata entry.

9.2.6.3 MPAI_AIFU_Resource_GetAIW

error_t MPAI_AIFU_Resource_GetAIW(int *AIW_ID*, const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the User Agent interrogates the resource allocation for one AIM Metadata entry for the AIW with AIW ID *AIW_ID*.

9.2.6.4 MPAI_AIFU_Resource_SetAIW

error_t MPAI_AIFU_Resource_SetAIW(int *AIW_ID*, const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the User Agent interrogates the resource allocation for one AIM Metadata entry for the AIW with AIW ID *AIW_ID*.

9.3 Controller API called by AIMs

9.3.1 General

The following API have been defined in Version 1.1. They specify how AIWs:

1. Define the topology and connections of AIMs in the AIW.
2. Define the Time base.
3. Define the Resource Policy.

9.3.2 Resource allocation management

9.3.2.1 MPAI_AIFM_Resource_GetGlobal

error_t MPAI_AIFM_Resource_GetGlobal(const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the AIM interrogates the resource allocation for one AIF Metadata entry.

9.3.2.2 MPAI_AIFM_Resource_SetGlobal

error_t MPAI_AIFM_Resource_SetGlobal(const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the AIM initialises the resource allocation for one AIF Metadata entry.

9.3.2.3 MPAI_AIFM_Resource_GetAIW

error_t MPAI_AIFM_Resource_GetAIW(int *AIW_ID*, const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the AIM interrogates the resource allocation for one AIM Metadata entry for the AIW with AIW ID *AIW_ID*.

9.3.2.4 MPAI_AIFM_Resource_SetAIW

error_t MPAI_AIFM_Resource_SetAIW(int *AIW_ID*, const char* *key*, const char* *min_value*, const char* *max_value*, const char* *requested_value*)

With this function the AIM interrogates the resource allocation for one AIM Metadata entry for the AIW with AIW ID *AIW_ID*.

9.3.3 Register/deregister AIMs with the Controller

9.3.3.1 MPAI_AIFM_AIM_Register_Local

error_t MPAI_AIFM_AIM_Register_Local(const char* *name*)

With this function the AIM registers the AIM named *name* with the Controller. The AIM shall be defined in the AIM Metadata. An Implementation that can be run on the Controller shall have been downloaded from the Store together with the Metadata or be available in the AIM Storage after having been downloaded from the Store together with the Metadata.

9.3.3.2 MPAI_AIFM_AIM_Register_Remote

error_t MPAI_AIFM_AIM_Register_Remote(const char* *name*, const char* *uri*)

With this function the AIM registers the AIM named *name* with the Controller. The AIM shall be defined in the AIM Metadata. An implementation that can be run on the Controller shall have been downloaded from the Store together with the Metadata or be available locally. The AIM will be run remotely on the MPAI Server identified by *uri*.

9.3.3.3 MPAI_AIFM_AIM_Deregister

error_t MPAI_AIFM_AIM_Deregister(const char* *name*)

The AIW deregisters the AIM named *name* from the Controller.

9.3.4 Send Start/Pause/Resume/Stop Messages to other AIMs

AIMs can send Messages to AIMs defined in its Metadata.

Errors encountered while transmitting/receiving these Messages are non-recoverable – i.e., they terminate the entire AIM. AIMs can communicate with other AIMs and the Controller uses this API to Start/Pause/Resume/Stop the AIMs.

9.3.4.1 MPAI_AIFM_AIM_Start

error_t MPAI_AIFM_AIM_Start(*const char* name*)

With this function the AIM asks the Controller to start the AIM named name. If the operation succeeds, it has immediate effect.

9.3.4.2 MPAI_AIFM_AIM_Pause

error_t MPAI_AIFM_AIM_Pause(*const char* name*)

With this function the AIM asks the Controller to pause the AIM named name. If the operation succeeds, it has immediate effect.

9.3.4.3 MPAI_AIFM_AIM_Resume

error_t MPAI_AIFM_AIM_Resume(*const char* name*)

With this function the AIM asks the Controller to resume the AIM named name. If the operation succeeds, it has immediate effect.

9.3.4.4 MPAI_AIFM_AIM_Stop

error_t MPAI_AIFM_AIM_Stop(*const char* name*)

With this function the AIM asks the Controller to stop the AIM named name. If the operation succeeds, it has immediate effect.

9.3.4.5 MPAI_AIFM_AIM_EventHandler

error_t MPAI_AIFM_AIM_EventHandler(*const char* name*)

The AIF creates EventHandler for the AIW with given name name. If the operation succeeds, it has immediate effect.

9.3.5 Register Connections between AIMs

9.3.5.1 MPAI_AIFM_Channel_Create

error_t
MPAI_AIFM_Channel_Create(const char* *name*, const char* *out_AIM_name*, const char* *out_port_name*, const char* *in_AIM_name*, const char* *in_port_name*)

With this function the AIM asks the Controller to create a new interconnecting channel between an output port and an input port. AIM and port names are specified with the name used when constructed.

9.3.5.2 MPAI_AIFM_Channel_Destroy

error_t MPAI_AIFM_Channel_Destroy(const char* *name*)

With this function the AIM asks the Controller to destroy the channel with name *name*. This API Call closes all Ports related to the Channel.

9.3.6 Using Ports

9.3.6.1 MPAI_AIFM_Port_Output_Read

message_t* MPAI_AIFM_Port_Output_Read(
const char* *AIM_name*, const char* *port_name*)

This function reads a message from the Port identified by (*AIM_name*,*port_name*). The read is blocking. Hence, in order to avoid deadlocks, the Implementation should first probe the Port with MPAI_AIF_Port_Probe. It returns a copy of the original Message.

9.3.6.2 MPAI_AIFM_Port_Input_Write

error_t MPAI_AIFM_Port_Input_Write(
const char* *AIM_name*, const char* *port_name*, message_t* *message*)

This function writes a message *message* to the Port identified by (*AIM_name*,*port_name*). The write is blocking. Hence, in order to avoid deadlocks the Implementation should first probe the Port with MPAI_AIF_Port_Probe. The Message being transmitted shall remain available until the function returns, or the behaviour will be undefined.

9.3.6.3 MPAI_AIFM_Port_Reset

error_t MPAI_AIFM_Port_Reset(const char* *AIM_name*, const char* *port_name*)

This function resets an input or output Port identified by (*AIM_name*,*port_name*) by deleting all the pending Messages associated with it.

9.3.6.4 MPAI_AIFM_Port_CountPendingMessages

size_t MPAI_AIFM_Port_CountPendingMessages(
const char* *AIM_name*, const char* *port_name*)

This function returns the number of pending messages on a input or output Port identified by (*AIM_name*,*port_name*).

9.3.6.5 MPAI_AIFM_Port_Probe

error_t MPAI_AIFM_Port_Probe(const char* *port_name*, message_t* *message*)

This function returns MPAI_AIF_OK if either the Port is a FIFO input port and an AIM can write to it, or the Port is a FIFO output Port and data is available to be read from it.

9.3.6.6 MPAI_AIFM_Port_Select

int MPAI_AIFM_Port_Output_Select(
const char* *AIM_name_1*,const char* *port_name_1,…*)

Given a list of output Ports, this function returns the index of one Port for which data has become available in the meantime. The call is blocking to address potential race conditions.

9.3.7 Operations on messages

All implementations shall provide a common Message passing functionality which is abstracted by the following functions.

9.3.7.1 MPAI_AIFM_Message_Copy

message_t* MPAI_AIFM_Message_Copy(message_t* *message*)

This function makes a copy of a Message structure *message*.

9.3.7.2 MPAI_AIFM_Message_Delete

message_t* MPAI_AIFM_Message_Delete(message_t* *message*)

This function deletes a Message *message* and its allocated memory. The format of each Message passing through a Channel is defined by the Metadata for that Channel.

9.3.7.3 MPAI_AIFM_Message_GetBuffer

void* MPAI_AIFM_Message_GetBuffer(message_t* *message*)

This function gets access to the low-level memory buffer associated with a message structure *message*.

9.3.7.4 MPAI_AIFM_Message_GetBufferLength

size_t MPAI_AIFM_Message_GetBufferLength(message_t* *message*)

This function gets the size in bits of the low-level memory buffer associated with a message structure *message*.

9.3.7.5 MPAI_AIFM_Message_Parse

parser_t* MPAI_AIFM_Message_Parse (const char* *type*)

This function creates a parsed representation of the data type defined in *type* according to the Metadata syntax defined in Subsection 6.1.1 Type system, to facilitate the successive parsing of raw memory buffers associated with message structures (see functions below).

9.3.7.6 MPAI_AIFM_Message_Parse_Get_StructField

void* MPAI_AIFM_Message_Parse_Get_StructField(
parser_t* *parser*, void* *buffer*, const char* *field_name*)

This function assumes that the low-level memory buffer *buffer* contains data of type struct_type whose complete parsed type definition (specified according to the metadata syntax defined in Subsection 6.1.1 Type system) can be found in *parser*. This function fetches the element of the struct_type named *field_name*, and return it in a freshly allocated low-level memory buffer. If a element with such name does not exist, return NULL.

9.3.7.7 MPAI_AIFM_Message_Parse_Get_VariantType

void* MPAI_AIFM_Message_Parse_Get_VariantType(
parser_t* *parser*, void* *buffer*, const char* *type_name*)

This function assumes that the low-level memory buffer *buffer* contains data of type variant_type whose complete parsed type definition (specified according to the Metadata syntax defined in Chapter 0, Type system) can be found in *parser*. Fetch the member of the variant_type named *field_name*, and return it in a freshly allocated low-level memory buffer. If a element with such name does not exist, return NULL.

9.3.7.8 MPAI_AIFM_Message_Parse_Get_ArrayLength

int MPAI_AIFM_Message_Parse_Get_ArrayLength(parser_t* *parser*, void* *buffer*)

This function assumes that the low-level memory buffer *buffer* contains data of type array_type whose complete parsed type definition (specified according to the Metadata syntax defined in Chapter Type system6.1.1, Type system) can be found in *parser*. Retrieve the length of such an array. If the buffer does not contain an array, return -1.

9.3.7.9 MPAI_AIFM_Message_Parse_Get_ArrayField

void* MPAI_AIFM_Message_Parse_Get_ArrayField(
parser_t* *parser*, void* *buffer*, const int *field_num*)

This function assumes that the low-level memory buffer *buffer* contains data of type array_type whose complete parsed type definition (specified according to the metadata syntax defined in Subsection 6.1.1, Type system) can be found in *parser*. Fetch the element of the array_type named *field_num*, and return it in a freshly allocated low-level memory buffer. If such element does not exist, return NULL.

9.3.7.10 MPAI_AIFM_Message_Parse_Delete

void MPAI_AIFM_Message_Parse_Delete(parser_t* *parser*)

This function deletes the parsed representation of a data type defined by *parser*, and deallocates all memory associated to it.

9.3.8 Functions specific to machine learning

The two key functionalities supported by the Framework are reliable update of AIMs with Machine Learning functionality and hooks for Explainability.

9.3.8.1 Support for model update

The following API supports AIM ML model update. Such update occurs via the Store by using the Store specific APIs or via Shared (SharedStorage) or AIM-specific (AIMStorage) storage by using the specified APIs.

error* MPAI_AIFM_Model_Update(const char* *model_name*)

The URI *model_name* points to the updated model. In some cases, such update needs to happen in highly available way so as not to impact the operation of the system. How this is effected is left to the Implementer.

9.3.8.2 Support for model drift

With this function the Controller detects possible degradation in ML operation caused by the characteristics of input data being significantly different from those used in training.

float MPAI_AIFM_Model_Drift(const char* *name*)

9.3.9 Controller API called by Controller

This Section specifies functions used by an AIM to Communicate through a Remote Port with an AIM running on another Controller. The local and remote AIMs shall belong to the same type of AIW.

9.3.9.1 MPAI_AIFM_External_List

error_t MPAI_AIFM_External_List(int* *num_in_range*, const char** *controllers_metadata*)

This function returns the number *num_in_range* of in-range Controllers with which it is possible to establish communication and running the same type of AIW, and a vector *controllers_metadata* containing AIW Metadata for each reachable Controller specified according to the JSON format defined in Section 6.3. In case more than one AIW of the same type is running on the same remote Controller, each such AIW is presented as a separate vector element.

9.3.9.2 MPAI_AIFM_External_Output_Read

message_t* MPAI_AIFM_External_Output_Read(int *controllerID*, const char* *AIM_name*, const char* *port_name*)

This function attempts to read a message from the External Port identified by (*controllerID, AIM_name*,*port_name*). The read is blocking. Hence, to avoid deadlocks, the Implementation should first probe the Port with MPAI_AIF_Port_Probe. It returns a copy of the original Message. This function attempts to establish a connection between the Controller and the external in-range Controller identified with a previous call to MPAI_AIFM_Communication_List. The call might fail due to the Controller not being in range anymore or other communication-related issues.

9.3.9.3 MPAI_AIFM_External_Input_Write

error_t MPAI_AIFM_External_Input_Write(int *controllerID,* const char* *AIM_name*, const char* *port_name*, message_t* *message*)

This function attempts to write a message *message* to the External Port identified by (*controllerID, AIM_name*, *port_name*). The write is blocking. Hence, in order to avoid deadlocks the Implementation should first probe the Port with MPAI_AIF_Port_Probe. The Message being transmitted shall remain available until the function returns, or the behaviour will be undefined. This function attempts to establish a connection between the Controller and the external in-range Controller identified with a previous call to MPAI_AIFM_Communication_List. The call might fail due to the Controller not being in range anymore or other communication-related issues.

10 Security API

10.1 Data characterisation structure

These API are intended to support developers who need a secure environment. They are divided into two parts: the first part includes APIs whose calls are executed in the non-secure area and the second part APIs whose calls that are executed in the secure area.
Data, independently from its usage (as a key, an encrypted payload, plain text, etc.) will be passed to/from the APIs through data_t structure.

The data_t structure shall include the following fields:

- data_location_t location

the identifier of the location of the data (see data_location_t below).

- void* data

the pointer (within the location specified above) to the start of the data/

- size_t size

the size of the data (in bytes).

- data_flags_t flags

other flags characterizing data.
The data_location_t is uint32_t type and can take one of the following symbolic values:

- DATA_LOC_RAM
- DATA_LOC_EXT_FLASH
- DATA_LOC_INT_FLASH
- DATA_LOC_LOCAL_DISK
- DATA_LOC_REMOTE_DISK

The data_flags_t is uint32_t type and can take one of the following symbolic values:

- DATA_FLAG_Encrypted
- DATA_FLAG_plain
- DATA_FLAG_UNKNOWN

10.2 API called by User Agent

User Agents calls Connect to Controller API

error_t MPAI_AIFU_Controller_Initialize_Secure(bool useAttestation)

This function, called by the User Agent, switches on and initialises the Controller, in particular the Secure Communication Component.

- Start AIW
- Suspend
- Resume
- Stop

10.3 API to access Secure Storage

In the following stringname is a symbolic name of the security memory area.

10.3.1 User Agent initialises Secure Storage API

Error_t MPAI_AIFSS_Storage_Init(string_t stringname, size_t data_length, const p_data_t data, flags_t flags flags)
Flags specify the initialisation behaviour.

10.3.2 User Agent writes Secure Storage API

Error_t MPAI_AIFSS_Storage_Write(string_t stringname, size_t data_length, const p_data_t data, flags_t flags flags)
Flags specify the write behaviour.

10.3.3 User Agent reads Secure Storage API

Error_t MPAI_AIFSS_Storage_Read(string_t stringname, size_t data_length,const p_data_t data, flags_t flags flags)
Flags specify the read behaviour.

10.3.4 User Agent gets info from Secure Storage API

Error_t MPAI_AIFSS_Storage_Getinfo(string_t stringname, struct storage_info_t * p_info)

10.3.5 User Agent deletes a p_data in Secure Storage API

Error_t MPAI_AIFSS_Storage_Delete(string_t stringname)
We assume that there is a mechanism that takes a stringname of type string and maps to a numeric uid

10.4 API to access Attestation

Controller Trusted Service Attestation call (part of the Trusted Services API)

Error_t MPAI_AIFAT_Get_Token(uint8_t *token_buf, size_t token_buf_size,size_t *token_size)

Token Buffer and Token Manage are managed by the code of the API implementation.
Based on CBOR [13], COSE [14] and EAT [15] standards.

10.5 API to access cryptographic functions

10.5.1 Hashing

There are many different hashing algorithms in use today, but some of the most common ones include:

- SHA (Secure Hash Algorithm) [24]: A family of hash functions developed by the US National Security Agency (NSA). The most widely used members of this family are SHA-1 and SHA-256, both of which are commonly used to generate digital signatures and verify data integrity.
- MD5 (Message-Digest Algorithm 5) [17]: A widely used hash function that produces 128-bit hash values. Although it is widely used, it is not considered secure and has been replaced by more secure hash functions in many applications.
-

Hash_state_t state object type

Implementation dependent

Error_t MPAI_AIFCR_Hash(Hash_state_t * state, algorithm_t alg, const uint8_t * hash, size_t * hash_length, size_t hash_size, const uint8_t * input, size_t input_length)
Perform a hash operation on an input data buffer producing the resulting hash in an output buffer. The encryption engine provides support for encryption/decryption of data of arbitrary size by processing it either in one chunk or multiple chunks. Implementation note: encryption engine should be efficient and release control to the rest of the system on a regular basis (e.g., at the end of a chunk computation).

Error_t MPAI_AIFCR_Hash_verify(Hash_state_t * state, const uint8_t * hash, size_t hash_length, const uint8_t * input, size_t input_length)

Perform a hash verification operation checking the hash against an input buffer.

Error_t MPAI_AIFCR_Hash_abort(Hash_state_t * state)

Abort operation and release internal resources.

10.5.2 Key management

Description:

- Applications **access keys indirectly** via an identifier
- Operations performed using a key **without accessing the key material**

If a key is externally provided it needs to map to the format below.
The key data is organised in a data structure that includes identifiers, the data itself, and the type of data as indicated below. The p_data structure includes information regarding the location where the key is stored.

10.5.2.1 MPAI_AIFKM_attributes_t structure

- Identifier **(number)**
- p_data (**structure)**
- Type:
    - RAW_DATA (none)
    - HMAC (hash)
    - DERIVE

        - PASSWORD (key derivation)
        - AES
        - DES
        - RSA (asymmetric RSA cipher)
        - ECC
        - DH (asymmetric DH key exchange).
- Lifetime
    - **persistence** level
    - **volatile** keys → lifetime AIF_KEY_LIFETIME_VOLATILE, stored in RAM
    - **persistent** keys → lifetime AIF_KEY_LIFETIME_PERSISTENT, stored in primary local storage or primary secure element.
- Policy
    - **set** of **usage flags** + permitted **algorithm**
    - permitted algorithms → restrict to a **single algorithm**, types: NONE or specific algorithm
    - usage flags → EXPORT, COPY, CACHE, ENCRYPT, DECRYPT, SIGN_MESSAGE, VERIFY_MESSAGE, SIGN_HASH, VERIFY_HASH, DERIVE, VERIFY_DERIVATION
    -

Error_t MPAI_AIFKM_import_key(const key_attributes_t * attributes, const uint8_t * data, size_t data_length, key_id_t * key)

When importing a key as a simple binary value, it is the responsibility of the programmer to fill in the attributes data structure. The identifier inside the attributes data structure will be internally generated as a response to the API call.

Error_t MPAI_AIFKM_generate_key(const attributes_t * attributes, key_id_t * key)

Generate key randomly.

Error_t MPAI_AIFKM_copy_key(key_id_t source_key, const key_attributes_t * attributes, key_id_t * target_key)

Copy key randomly.

Error_t MPAI_AIFKM_destroy_key(key_id_t key)

Destroy key.

Error_t MPAI_AIFKM_export_key(key_id_t key, uint8_t * data, size_t data_size, size_t * data_length)

Export key to output buffer.

Error_t MPAI_AIFKM_export_public_key(key_id_t key, uint8_t * data, size_t data_size, size_t * data_length);

Export public key to output buffer.

10.5.3 Key exchange

Algorithms: FFDH (finite-field Diffie-Hellman) [20], ECDH (elliptic curve Diffie-Hellman) [23]

Error_t MPAI_AIFKX_raw_key_agreement(algorithm_t alg,key_id_t private_key,const uint8_t * peer_key,size_t peer_key_length,uint8_t * output,size_t output_size,size_t * output_length)

Return the raw shared secret.

Error_t MPAI_AIFKX_key_derivation_key_agreement(key_derivation_operation_t * operation,key_derivation_step_t step,key_id_t private_key,const uint8_t * peer_key,size_t peer_key_length)

Key agreement and use the shared secret as input to a key derivation.

10.5.4 Message Authentication Code

The code is a cryptographic checksum on data. It uses a session key with the goal to detect any modification of the data. It requires the data and the shared session key known to the data originator and recipients. The cryptographic algorithms of algorithm_t are the same as defined above.

mac_state_t
Implementation dependent.

error_t MPAI_AIFMAC_sign_setup(mac_state_t * state, key_id_t key, algorithm_t alg)

Setup MAC **sign** operation.

error_t MPAI_AIFMAC_verify_setup(mac_state _t * state, key_id_t key, algorithm_t alg)

Setup MAC **verify** operation.

error_t MPAI_AIFMAC_update(mac_state_t * state, const uint8_t * input, size_t input_length)

Compute MAC for a chunk of data (can be repeated several times until completion of data).

error_t MPAI_AIFMAC_mac_sign_finish(mac_state_t * state, uint8_t * mac, size_t mac_size, size_t * mac_length)

Finish MAC **sign** operation.

error_t MPAI_AIFMAC_mac_verify_finish(mac_state_t * state, const uint8_t * mac, size_t mac_length)

Finish MAC **verify** operation at receiver side.

error_t MPAI_AIFMAC_mac_abort(mac_state_t * state)

Abort MAC operation.

10.5.5 Cyphers

MPAI-AIF V2 assumes that, in case multiblock cipher is used, the developer shall manage the IV parameter by explicitly generating the IV, i.e.:

1. Not relying on a service doing that for them.
2. Securely communicating the IV to the parties receiving the message, and
3. If the IV is not disposed of, storing the IV in the Secure Storage.

Algorithms: AIF_ALG_XTS [16], AIF_ALG_ECB_NO_PADDING [25], AIF_ALG_CBC_NO_PADDING [25], AIF_ALG_CBC_PKCS7 [25]
In the following API calls, the IV parameter and IV size shall be set to NULL if not needed by the specific call. An IV shall securely generated by the API implementation in case the encryption algorithm needs an IV and NULL is passed to the API.

cipher_state_t
State object type (implementation dependent). In future version the state type may be defined.

Error_t MPAI_AIFCIP_Encrypt(cipher_state_t * state, key_id_t key, algorithm_t alg, uint8_t * iv, size_t iv_size, size_t * iv_length)

Setup symmetric encryption.

Error_t MPAI_AIFCIP_Decrypt(cipher_state_t * state, key_id_t key, algorithm_t alg, uint8_t * iv, size_t iv_size, size_t * iv_length)

Setup symmetric decryption.

Error_t MPAI_AIFCIP_Abort(cipher_state_t * state)

Abort symmetric encryption/decryption.

10.5.6 Authenticated encryption with associated data (AEAD)

Algorithms: ALG_GCM [26], ALG_CHACHA20_POLY1305 [19].

PSA_ALG_GCM **requires a nonce** of at least 1 byte in length.
aead_state_t
state object type (implementation dependent). In future version the state type may be defined.

Error_t MPAI_AIFAEAD_Encrypt(aead_state_t * state, key_id_t key, algorithm_t alg, const uint8_t * nonce, size_t nonce_length, const uint8_t * additional_data, size_t additional_data_length, const uint8_t * plaintext, size_t plaintext_length, uint8_t * ciphertext, size_t ciphertext_size, size_t * ciphertext_length)

Error_t MPAI_AIFAEAD_Decrypt(aead_state_t * state, key_id_t key, algorithm_t alg, const uint8_t * nonce, size_t nonce_length, const uint8_t * additional_data, size_t additional_data_length, const uint8_t * ciphertext, size_t ciphertext_length, uint8_t * plaintext, size_t plaintext_size, size_t * plaintext_length)

Error_t MPAI_AIFEAD_Abort(aead_state_t * state)

10.5.7 Signature

Algorithms: RSA_PKCS1V15_SIGN [21], RSA_PSS [21], ECDSA [18], PURE_EDDSA [22].
sign_state_t
State object type (implementation dependent). In future version the state type may be defined.

Error_t MPAI_AIFSIGN_sign_message(sign_state_t * state, key_id_t key, algorithm_t alg, const uint8_t * input, size_t input_length, uint8_t * signature, size_t signature_size, size_t *signature_length)
Sign a message with a private key (for hash-and-sign algorithms, this includes the hashing step).

Error_t MPAI_AIFSIGN_verify_message(sign_state_t * state, key_id_t key, algorithm_t alg, const uint8_t * input, size_t input_length, const uint8_t * signature, size_t signature_length)
Verify a signature with a public key (for hash-and-sign algorithms, this includes the hashing step).
psa_status_t psa_sign_hash(psa_key_id_t key, psa_algorithm_t alg, const uint8_t * hash, size_t hash_length, uint8_t * signature, size_t signature_size, size_t * signature_length)
Sign an already-calculated hash with a private key.
psa_status_t psa_verify_hash(psa_key_id_t key, psa_algorithm_t alg, const uint8_t * hash, size_t hash_length, const uint8_t * signature,
size_t signature_length)
Verify the signature of a hash.

10.5.8 Asymmetric Encryption

Algorithms: RSA_PKCS1V15_CRYPT [21], RSA_OAEP [21].

psa_status_t psa_asymmetric_encrypt(psa_key_id_t key, psa_algorithm_t alg, const uint8_t * input, size_t input_length, const uint8_t * salt, size_t salt_length, uint8_t * output, size_t output_size, size_t * output_length)
Encrypt a short message with a public key.

psa_status_t psa_asymmetric_decrypt(psa_key_id_t key, psa_algorithm_t alg, const uint8_t * input, size_t input_length, const uint8_t * salt, size_t salt_length, uint8_t * output, size_t output_size, size_t * output_length)

Decrypt a short message with a private key.

10.6 API to enable secure communication

An implementer should rely on the CoAP and HTTPS support provided by secure transport libraries for the different programming languages.

11 Profiles

11.1 Basic Profile

The Basic Profile utilises:

1. Non-Secure Controller.
2. Non-Secure Storage.
3. Secure Communication enabled by secure communication libraries.
4. Basic API.

11.2 Secure Profile

Uses all the technologies in this Technical Specification.

12 Data Types

MPAI-AIF V2-1 specifies one Data Type:

Machine Learning Model

13 Examples

13.1 AIF Implementations

This Chapter contains informative examples of high-level descriptions of possible AIF operations. This Chapter will continue to be developed in subsequent Version of this Technical Specification by adding more examples.

13.1.1 Resource-constrained implementation

1. Controller is a single process that implements the AIW and operates based on interrupts call-backs.
2. AIF is instantiated via a secure communication interface.
3. AIMs can be local or has been instantiated through a secure communication interface.
4. Controller initialises the AIF.
5. AIF asks the AIMs to be instantiated.
6. Controller manages the Events and Messages.
7. User Agent can act on the AIWs at the request of the user.

13.1.2 Non-resource-constrained implementation

1. Controller and AIW are two independent processes.
2. Controller manages the Events and Messages.
3. AIW contacts Controller on Communication and authenticates itself.
4. Controller requests AIW configuration metadata.
5. AIW sends Controller the configuration metadata.
6. The implementation of the AIW can be local or can be downloaded from the MPAI Store.
7. Controller authenticates itself with the MPAI Store and requests implementations for the needed AIMs listed in the metadata from the MPAI Store.
8. The Store sends the requested AIM implementations and the configuration metadata.
9. Controller:
    1. Instantiates the AIMs specified in the AIW metadata.
    2. Manages their communication and resources by sending Messages to AIMs.
10. User Agent can gain control of AIWs running on the Controller via a specific Controller API, e.g., User Agent can test conformance of a AIW with an MPAI standard through a dedicated API call.

13.2 Examples of types

byte[] bitstream_t

An array of bytes, with variable length.

{int32 frameNumber; int16 x; int16 y; byte[] frame} frame_t

A struct_type with 4 members named frameNumber, x, y, and frame — they are an int32, an int16, an int16, and an array of bytes with variable length, respectively.

{int32 i32 | int64 i64} variant_t

A variant_type that can be either an int32 or an int64.

13.3 Examples of Metadata

This section contains the AIF, AIW and AIM Metadata of the Enhanced Audioconference Experience (CAE-EAE) V2.1 as examples.

13.3.1 Enhanced Audioconference Experience AIF

https://schemas.mpai.community/AIF/V2.0/AIF-metadata.schema.json

13.3.2 Enhanced Audioconference Experience AIW

https://schemas.mpai.community/CAE/V2.1/AIWs/EnhancedAudioconferenceExperience.json

13.3.3 Analysis Transform AIM

https://schemas.mpai.community/CAE/V2.1/AIMs/AudioAnalysisTransform.json

13.3.4 Sound Field Description AIM

https://schemas.mpai.community/CAE/V2.1/AIMs/SoundFieldDescription.json

13.3.5 Speech Detection and Separation AIM

https://schemas.mpai.community/CAE/V2.1/AIMs/SpeechDetectionandSeparation.json

13.3.6 Noise Cancellation Module AIM

https://schemas.mpai.community/CAE/V2.1/AIMs/NoiseCancellationModule.json

13.3.7 Audio Synthesis Transform AIM

https://schemas.mpai.community/CAE/V2.1/AIMs/AudioSynthesisTransform.json

13.3.8 Audio Description Packaging AIM

https://schemas.mpai.community/CAE/V2.1/AIMs/AudioDescriptionPackaging.json