Document Title	Demonstrator Design of Functional Cluster Identity and Access Management
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	901

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R20-11

Document Change History

Date	Release	Changed by	Description
2020-11-30	R20-11	AUTOSAR Release Management	• Rework of the Access Manager and removal of capabilities • Introduction of separate grant for demonstrational purposes • Usage of the execution manager to identify processes
2019-11-28	R19-11	AUTOSAR Release Management	• Dependency to EM internal parser replaced with ara::per API • Removed generated API tables • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• No functional changes
2018-11-02	18-10	AUTOSAR Release Management	• Code and CMake files reworked to follow coding guidelines • Multiple internal code improvements • Implement Docker-based unit testing approach

2018-05-02	18-03	AUTOSAR Release Management	• Use Unix Domain Sockets instead of the FIFO-based libplatform • Demonstrate credential passing over sockets • Introduction of configurable policies parsed and evaluated in the Access Manager • Refactoring into seperate FC and build system overhaul
2017-11-22	17-10	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction

This document describes the design (approach and decisions) of the Functional Cluster Identity and Access Management for AUTOSAR's Adaptive Platform Demonstrator.

The decisions taken for the AUTOSAR Adaptive Platform Demonstrator may not apply to other implementations. Nevertheless, the Demonstrator may supplement and ease the understanding of the specifications.

This implementation provides a reference implementation that is meant to be used in prototype projects as well as a basis for production-grade implementations. The main goal is to provide proof of concepts for identity and access management features specified in AUTOSAR Adaptive. Optimizations for memory consumption and speed have been considered but they haven't played a major role in the overall software design and implementation. Any series production project deriving from this implementation will have to further fulfill the safety constraints described in the industry standards.

1.1 Purpose and functionality

The Identity and Access Management offers grant query functionality for AUTOSAR Adaptive Platform.

A central instance, the **Access Manager** (2.2.4), collects the grant information that is stored in a Manifest file on the system.

The **GrantQueryClient** (2.3.1) provides methods to ease the implementation of Policy Enforcement Points (PEPs) in other functional clusters or services. It implements an IPC connection to the **Access Manager** (2.2.4) to check if a specific grant has been allowed.

The implementation of ara:iam contains the access management aspects of the functional cluster. A demo functional cluster (fc) has been implementend as part of the sample-applications (subfolder sec_examples/demo_fc) to show a possible implementation of a PEP.

To identify the caller of a function from a functional cluster is the responsibility of the implementation of the PEP in the functional cluster itself. One possibility to identify a caller of a functional cluster could be the usage of unix domain sockets to communicate between the adaptive application (caller) and the functional cluster. This allows to obtain the process id (pid) of the caller. A possible implementation can found in the ipc library provided by iam. In combination with the process lookup API by the execution manager it is possible to map the pid to the calling application. It is up to the fc implementation to ensure that the (original) caller is still running and that the pid has not been reassigned to a different application during the duration of the lookup.

1.2 Implementation status

An implementation of the **Access Manager** (2.2.4) has been created. The **Access Manager** (2.2.4) can read simple a json file containing the grants and answer IPC queries for individual requests. An **API** (2.3.1) has been implemented to provide easy access to the grant information, hiding the technical details of the IPC communication from the clients.

1.3 Known limitations

This is the list of limitations the reference implementation currently has:

- *Only demo grant available.* Currently one grant for demonstration and testing purposes is available.
- *Grants specified by one single json file.* Currently grants need to be specified by one single json file. This json file can currently not be generated from an arxml file.

1.4 Deviations from specification

No documented deviations.

2 Overview on architecture

2.1 Design approach / Design principles

2.1.1 Overview

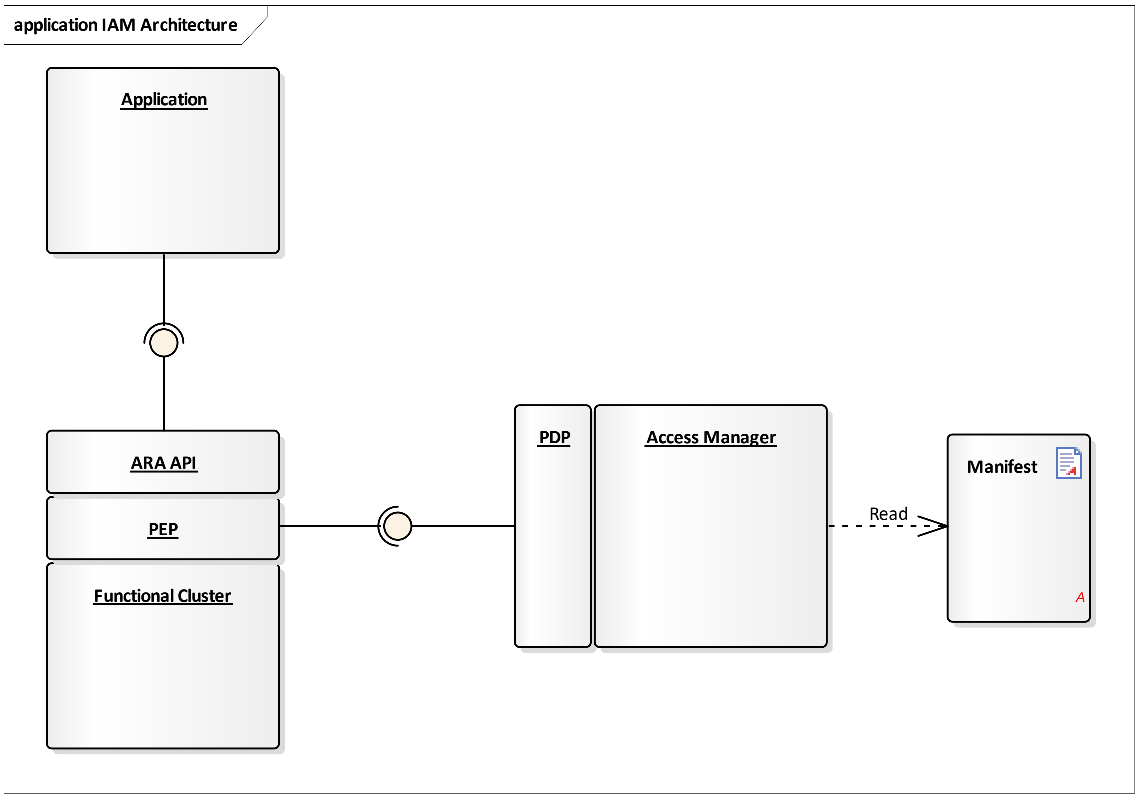

Figure 2.1: Architectural overview of the access management

A system application, the Access Manager (AM), centrally gathers the grant information and provides an ipc interface for queries of that information. The Access Manager thus represents a central Policy Decision Point (PDP).

Policy Enforcement Points (PEPs) lie within the functional clusters and services that offer functionality. Using the GrantyQueryClient API, the code implementing a PEP can perform ipc queries to the Access Manager to determine whether a given action should be permitted.

2.1.2 Testability

To allow for an implementation of unit tests with as little side effects and dependencies as possible, many classes are split into separate interface and implementation (Impl) classes, and foreign classes that do not follow this separation are wrapped into custom adapter classes.

To simplify the internals of the iam implementation the unit tests as well as helper classes for the unit test are not shown in the class diagrams of this chapter.

2.2 Class Overview

2.2.1 IPC

The IAM implementation provides its own library for ipc communication.

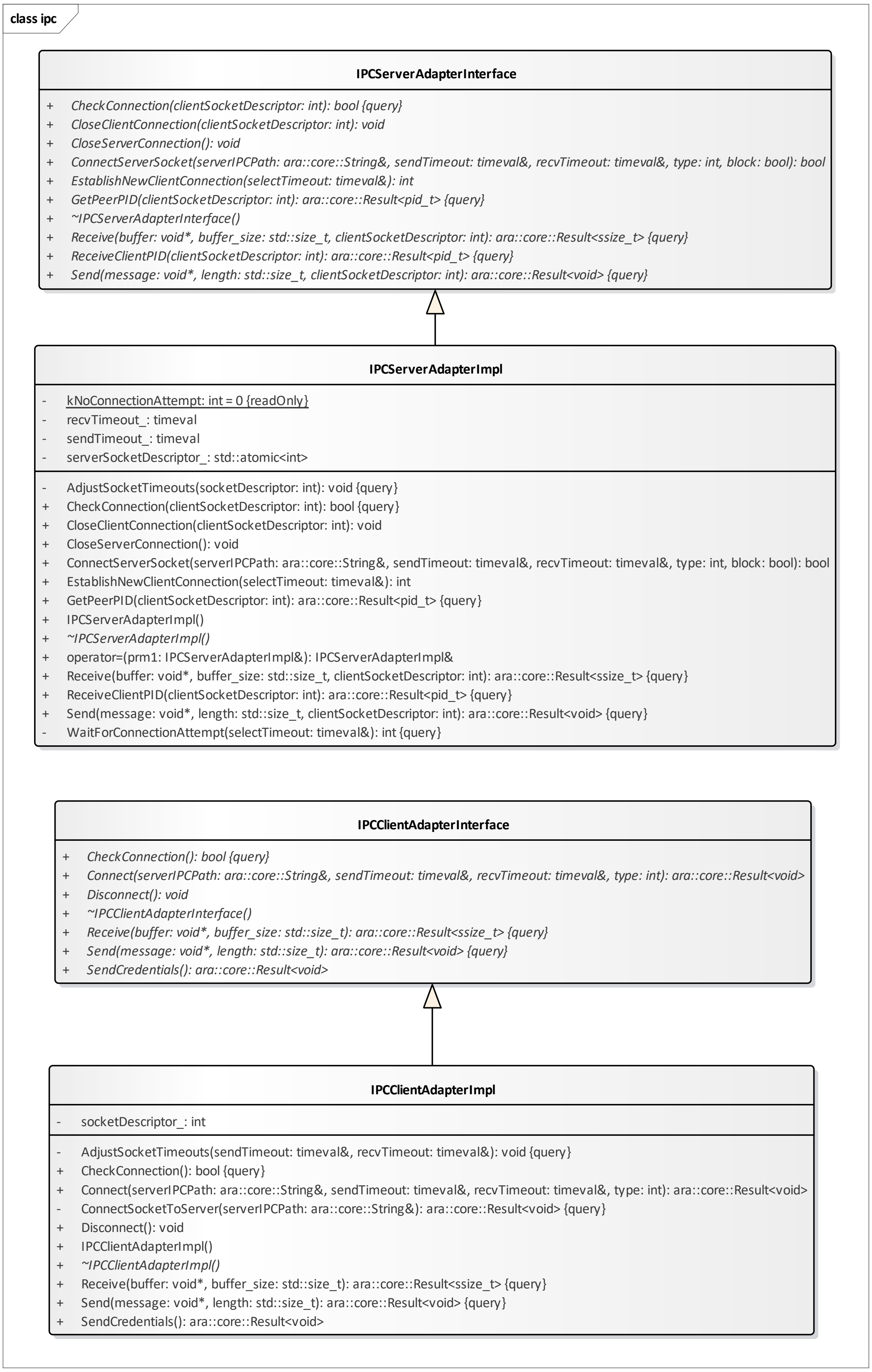
class ipc
IPCServerAdapterInterface
+ CheckConnection(clientSocketDescriptor: int): bool {query}
+ CloseClientConnection(clientSocketDescriptor: int): void
+ CloseServerConnection(): void
+ ConnectServerSocket(serverIPCPath: ara::core::String&, sendTimeout: timeval&, recvTimeout: timeval&, type: int, block: bool): bool
+ EstablishNewClientConnection(selectTimeout: timeval&): int
+ GetPeerPID(clientSocketDescriptor: int): ara::core::Result<pid_t> {query}
+ ~IPCServerAdapterInterface()
+ Receive(buffer: void*, buffer_size: std::size_t, clientSocketDescriptor: int): ara::core::Result<ssize_t> {query}
+ ReceiveClientPID(clientSocketDescriptor: int): ara::core::Result<pid_t> {query}
+ Send(message: void*, length: std::size_t, clientSocketDescriptor: int): ara::core::Result<void> {query}
IPCServerAdapterImpl
- kNoConnectionAttempt: int = 0 {readOnly}
- recvTimeout_: timeval
- sendTimeout_: timeval
- serverSocketDescriptor_: std::atomic<int>
- AdjustSocketTimeouts(socketDescriptor: int): void {query}
+ CheckConnection(clientSocketDescriptor: int): bool {query}
+ CloseClientConnection(clientSocketDescriptor: int): void
+ CloseServerConnection(): void
+ ConnectServerSocket(serverIPCPath: ara::core::String&, sendTimeout: timeval&, recvTimeout: timeval&, type: int, block: bool): bool
+ EstablishNewClientConnection(selectTimeout: timeval&): int
+ GetPeerPID(clientSocketDescriptor: int): ara::core::Result<pid_t> {query}
+ IPCServerAdapterImpl()
+ ~IPCServerAdapterImpl()
+ operator=(prm1: IPCServerAdapterImpl&): IPCServerAdapterImpl&
+ Receive(buffer: void*, buffer_size: std::size_t, clientSocketDescriptor: int): ara::core::Result<ssize_t> {query}
+ ReceiveClientPID(clientSocketDescriptor: int): ara::core::Result<pid_t> {query}
+ Send(message: void*, length: std::size_t, clientSocketDescriptor: int): ara::core::Result<void> {query}
- WaitForConnectionAttempt(selectTimeout: timeval&): int {query}
IPCClientAdapterInterface
+ CheckConnection(): bool {query}
+ Connect(serverIPCPath: ara::core::String&, sendTimeout: timeval&, recvTimeout: timeval&, type: int): ara::core::Result<void>
+ Disconnect(): void
+ ~IPCClientAdapterInterface()
+ Receive(buffer: void*, buffer_size: std::size_t): ara::core::Result<ssize_t> {query}
+ Send(message: void*, length: std::size_t): ara::core::Result<void> {query}
+ SendCredentials(): ara::core::Result<void>
IPCClientAdapterImpl
- socketDescriptor_: int
- AdjustSocketTimeouts(sendTimeout: timeval&, recvTimeout: timeval&): void {query}
+ CheckConnection(): bool {query}
+ Connect(serverIPCPath: ara::core::String&, sendTimeout: timeval&, recvTimeout: timeval&, type: int): ara::core::Result<void>
- ConnectSocketToServer(serverIPCPath: ara::core::String&): ara::core::Result<void> {query}
+ Disconnect(): void
+ IPCClientAdapterImpl()
+ ~IPCClientAdapterImpl()
+ Receive(buffer: void*, buffer_size: std::size_t): ara::core::Result<ssize_t> {query}
+ Send(message: void*, length: std::size_t): ara::core::Result<void> {query}
+ SendCredentials(): ara::core::Result<void>

Figure 2.2: Class diagram: IPC

For a client ipc socket the class IPCClientAdapterImpl is used. This class inherits from an abstract base class IPCClientAdapterInterface.

For a server ipc socket the class IPCServerAdapterImpl is used. This class inherits from an abstract base class IPCServerAdapterInterface. The implementation also provides the functionality to obtain the process id of the client (by using GetPeerPID or by using ReceiveClientPID and ReceiveClientPID process.

The uses of abstract base classes for the client as well as the server allows mocking of the implementation of the ipc channel to allow further unit testing.

2.2.2 Grant

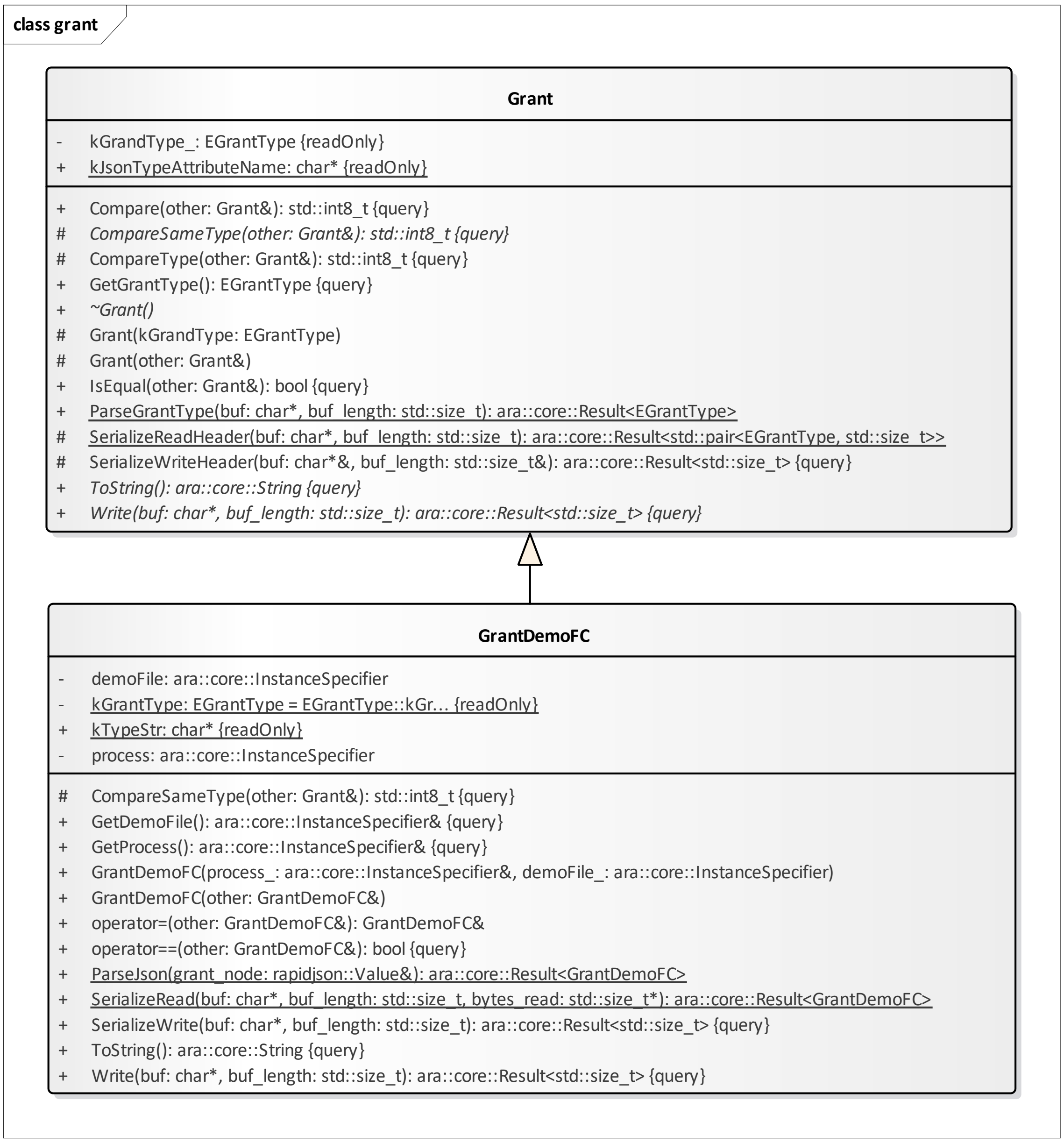
class grant
Grant
- kGrandType_: EGrantType {readOnly}
+ kJsonTypeAttributeName: char* {readOnly}
+ Compare(other: Grant&): std::int8_t {query}
# CompareSameType(other: Grant&): std::int8_t {query}
# CompareType(other: Grant&): std::int8_t {query}
+ GetGrantType(): EGrantType {query}
+ ~Grant()
# Grant(kGrandType: EGrantType)
# Grant(other: Grant&)
+ IsEqual(other: Grant&): bool {query}
+ ParseGrantType(buf: char*, buf_length: std::size_t): ara::core::Result<EGrantType>
# SerializeReadHeader(buf: char*, buf_length: std::size_t): ara::core::Result<std::pair<EGrantType, std::size_t>>
# SerializeWriteHeader(buf: char*&, buf_length: std::size_t&): ara::core::Result<std::size_t> {query}
+ ToString(): ara::core::String {query}
+ Write(buf: char*, buf_length: std::size_t): ara::core::Result<std::size_t> {query}
GrantDemoFC
- demoFile: ara::core::InstanceSpecifier
- kGrantType: EGrantType = EGrantType::kGr... {readOnly}
+ kTypeStr: char* {readOnly}
- process: ara::core::InstanceSpecifier
# CompareSameType(other: Grant&): std::int8_t {query}
+ GetDemoFile(): ara::core::InstanceSpecifier& {query}
+ GetProcess(): ara::core::InstanceSpecifier& {query}
+ GrantDemoFC(process_: ara::core::InstanceSpecifier&, demoFile_: ara::core::InstanceSpecifier)
+ GrantDemoFC(other: GrantDemoFC&)
+ operator=(other: GrantDemoFC&): GrantDemoFC&
+ operator==(other: GrantDemoFC&): bool {query}
+ ParseJson(grant_node: rapidjson::Value&): ara::core::Result<GrantDemoFC>
+ SerializeRead(buf: char*, buf_length: std::size_t, bytes_read: std::size_t*): ara::core::Result<GrantDemoFC>
+ SerializeWrite(buf: char*, buf_length: std::size_t): ara::core::Result<std::size_t> {query}
+ ToString(): ara::core::String {query}
+ Write(buf: char*, buf_length: std::size_t): ara::core::Result<std::size_t> {query}

Figure 2.3: Class diagram: Grant

If a grant is checking that a specific adaptive application has access to a resource, a reference (e.g. process short name) must be a member variable of the specialization of the abstract base class. Futher member variables can be added if the grant requires futher parameters to specify the action which needs to be checked.

The different child classes of Grant shall be using a different identifier value of the enum EGrantType. If two grants are using the same value of EGrantType the two grants shall be instances of the same class. Once a class has been given a specific EGrantType value the class should be marked as final to ensure that this class cannot be the base class of other grants.

Furthermore each child class must have a public static const data member kTypeStr which specifies the type string which is used as part of the json serialization of the current child class.

Additionally all the abstract functions of Grant must be implemented. These functions are needed to compare and sort different instances and serialization.

The serialization is not only needed to parse the grants given in the json format, but also to transfer the grants over an ipc channel. To allow a static allocation of the memory used to transfer grants the binary serialization of a grant may use up to kMaxSizeBytes GrantIPC bytes. If the binary serialization requires more bytes the value of kMaxSize BytesGrantIPC must be increased.

Code parts, which will require some changes if a new grant is added, are using static_ asserts no ensure that the needed code changes can be found very easy.

A possible implementation of a grant is the class GrantDemoFC, which is used for testing purposes.

2.2.3 GrantyQueryClient

The client library can be used to check if a grant is allowed. Grants are specified by child classes of the base class base Grant. The grant GrantDemoFC is used for testing purposes.

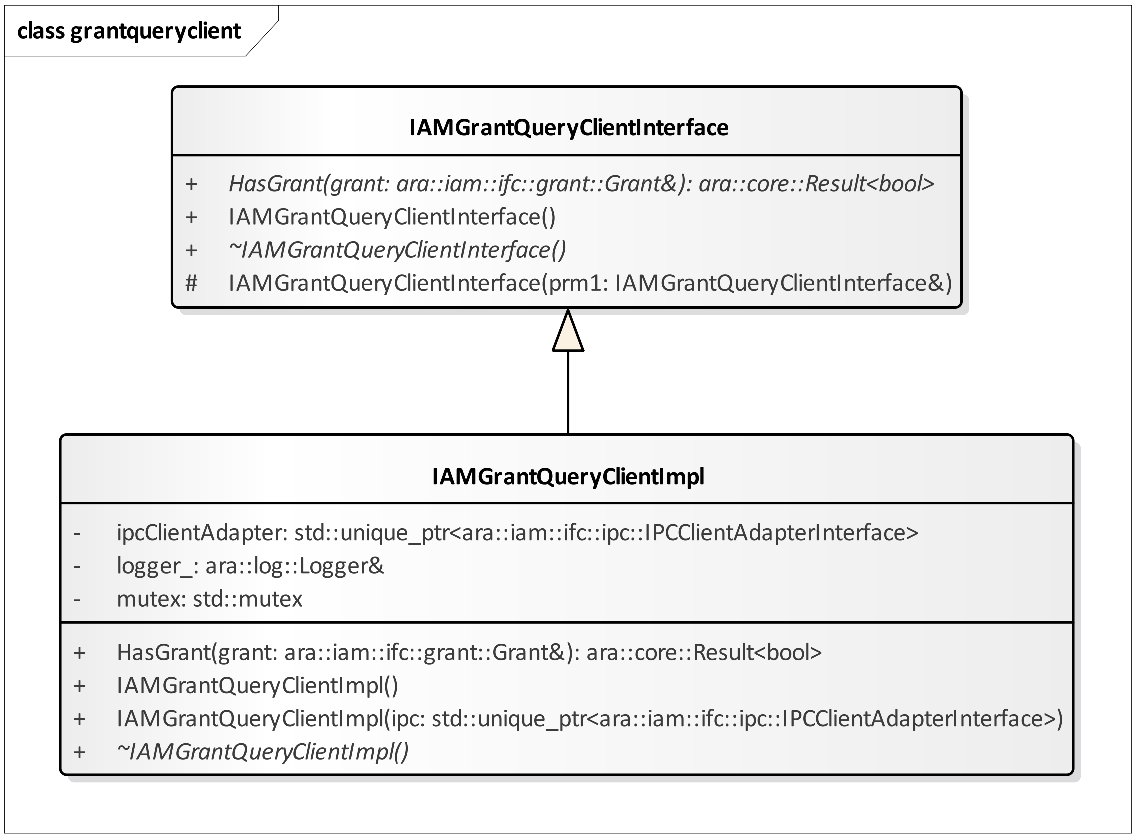
class grantqueryclient
IAMGrantQueryClientInterface
+ HasGrant(grant: ara::iam::ifc::grant::Grant&): ara::core::Result<bool>
+ IAMGrantQueryClientInterface()
+ ~IAMGrantQueryClientInterface()
# IAMGrantQueryClientInterface(prm1: IAMGrantQueryClientInterface&)
IAMGrantQueryClientImpl
- ipcClientAdapter: std::unique_ptr<ara::iam::ifc::ipc::IPCClientAdapterInterface>
- logger_: ara::log::Logger&
- mutex: std::mutex
+ HasGrant(grant: ara::iam::ifc::grant::Grant&): ara::core::Result<bool>
+ IAMGrantQueryClientImpl()
+ IAMGrantQueryClientImpl(ipc: std::unique_ptr<ara::iam::ifc::ipc::IPCClientAdapterInterface>)
+ ~IAMGrantQueryClientImpl()

Figure 2.4: Class diagram: GrantyQueryClient

The client library consists of a single function ara::iam::ifc::grantqueryclient::HasGrant<T> where the template parater T is the name of the class of the grant instance. The return value of the function is a ara::core::Future<bool>. If an error occurs and the request connot be processed an appropriate error code is returned.

This function creates a separate thread which an instance of the IAMGrantQueryClientImpl to communicate with the Access Manager via an ipc connection. For the ipc connection the class IPCClientAdapterImpl is used.

2.2.4 AccessManager

The IAM implementation is using a whitelist based approach. Grants stored in the manifest file used by the AccessManager specify allowed actions. Every action which is not specified is not allowed. The AccessManager acts as a central instance to process all incoming requests.

The AccessManager is realized as a central class with an event loop triggered by the method RunEventLoop, periodically calling ProcessGrantQueryRequests to read and process the grant requests coming in via an ipc channel. The ProcessGrantQueryRequests function checks if a new Request has been made and creates a separate Thread to process this request. For the ipc connection the class IPCServerAdapterImpl is used.

Each requests consist of a serialized child class of the base class Grant. To check if a grant has been given, the AccessManager performs a check if it can find an identical instance among the list of allowed grants. It that the received grants is not contained in the list of allowed grants the grant is forbidden.

Since the AccessManager spans the processing of in separate threads it is necessary to allow different threads to access its internal data. These members are stored in the struct AccessManagerData which is shared among the threads.

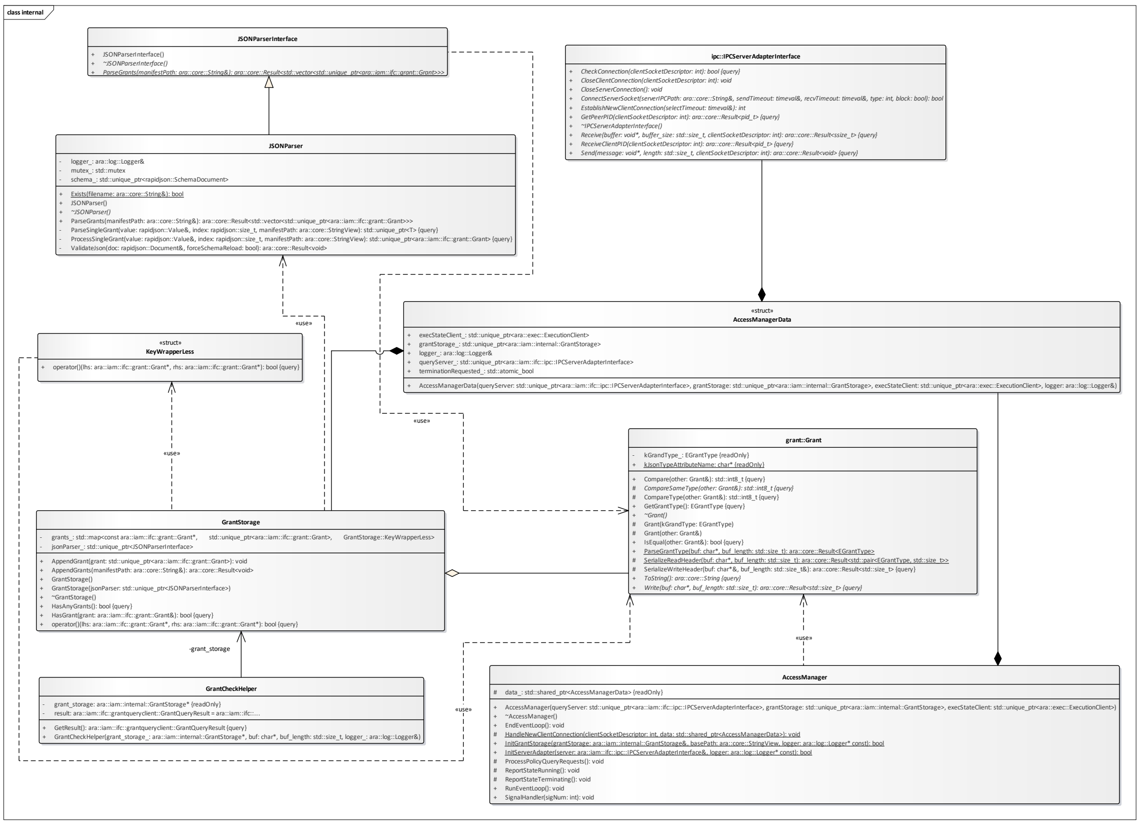
class internal
JSONParserInterface
+ JSONParserInterface()
+ ~JSONParserInterface()
+ ParseGrants(manifestPath: ara::core::String&): ara::core::Result<std::vector<std::unique_ptr<ara::iam::ifc::grant::Grant>>>
JSONParser
- logger_: ara::log::Logger&
- mutex_: std::mutex
- schema_: std::unique_ptr<rapidjson::SchemaDocument>
+ Exists(filename: ara::core::String&): bool
+ JSONParser()
+ ~JSONParser()
+ ParseGrants(manifestPath: ara::core::String&): ara::core::Result<std::vector<std::unique_ptr<ara::iam::ifc::grant::Grant>>>
- ParseSingleGrant(value: rapidjson::Value&, index: rapidjson::size_t, manifestPath: ara::core::StringView): std::unique_ptr<T> {query}
- ProcessSingleGrant(value: rapidjson::Value&, index: rapidjson::size_t, manifestPath: ara::core::StringView): std::unique_ptr<ara::iam::ifc::grant::Grant> {query}
- ValidateJson(doc: rapidjson::Document&, forceSchemaReload: bool): ara::core::Result<void>
ipc::IPCServerAdapterInterface
+ CheckConnection(clientSocketDescriptor: int): bool {query}
+ CloseClientConnection(clientSocketDescriptor: int): void
+ CloseServerConnection(): void
+ ConnectServerSocket(serverIPCPath: ara::core::String&, sendTimeout: timeval&, recvTimeout: timeval&, type: int, block: bool): bool
+ EstablishNewClientConnection(selectTimeout: timeval&): int
+ GetPeerPID(clientSocketDescriptor: int): ara::core::Result<pid_t> {query}
+ ~IPCServerAdapterInterface()
+ Receive(buffer: void*, buffer_size: std::size_t, clientSocketDescriptor: int): ara::core::Result<ssize_t> {query}
+ ReceiveClientPID(clientSocketDescriptor: int): ara::core::Result<pid_t> {query}
+ Send(message: void*, length: std::size_t, clientSocketDescriptor: int): ara::core::Result<void> {query}
«use»
«struct»
KeyWrapperLess
+ operator()(lhs: ara::iam::ifc::grant::Grant*, rhs: ara::iam::ifc::grant::Grant*): bool {query}
«struct»
AccessManagerData
+ execStateClient_: std::unique_ptr<ara::exec::ExecutionClient>
+ grantStorage_: std::unique_ptr<ara::iam::internal::GrantStorage>
+ logger_: ara::log::Logger&
+ queryServer_: std::unique_ptr<ara::iam::ifc::ipc::IPCServerAdapterInterface>
+ terminationRequested_: std::atomic_bool
+ AccessManagerData(queryServer: std::unique_ptr<ara::iam::ifc::ipc::IPCServerAdapterInterface>, grantStorage: std::unique_ptr<ara::iam::internal::GrantStorage>, execStateClient: std::unique_ptr<ara::exec::ExecutionClient>, logger: ara::log::Logger&)
«use»
«use»
grant::Grant
- kGrandType_: EGrantType {readOnly}
+ kJsonTypeAttributeName: char* {readOnly}
+ Compare(other: Grant&): std::int8_t {query}
# CompareSameType(other: Grant&): std::int8_t {query}
# CompareType(other: Grant&): std::int8_t {query}
+ GetGrantType(): EGrantType {query}
+ ~Grant()
# Grant(kGrandType: EGrantType)
# Grant(other: Grant&)
+ IsEqual(other: Grant&): bool {query}
+ ParseGrantType(buf: char*, buf_length: std::size_t): ara::core::Result<EGrantType>
# SerializeReadHeader(buf: char*, buf_length: std::size_t): ara::core::Result<std::pair<EGrantType, std::size_t>>
# SerializeWriteHeader(buf: char*&, buf_length: std::size_t&): ara::core::Result<std::size_t> {query}
+ ToString(): ara::core::String {query}
+ Write(buf: char*, buf_length: std::size_t): ara::core::Result<std::size_t> {query}
GrantStorage
- grants_: std::map<const ara::iam::ifc::grant::Grant*, std::unique_ptr<ara::iam::ifc::grant::Grant>, GrantStorage::KeyWrapperLess>
- jsonParser_: std::unique_ptr<JSONParserInterface>
+ AppendGrant(grant: std::unique_ptr<ara::iam::ifc::grant::Grant>): void
+ AppendGrants(manifestPath: ara::core::String&): ara::core::Result<void>
+ GrantStorage()
+ GrantStorage(jsonParser: std::unique_ptr<JSONParserInterface>)
+ ~GrantStorage()
+ HasAnyGrants(): bool {query}
+ HasGrant(grant: ara::iam::ifc::grant::Grant&): bool {query}
+ operator()(lhs: ara::iam::ifc::grant::Grant*, rhs: ara::iam::ifc::grant::Grant*): bool {query}
-grant_storage
«use»
GrantCheckHelper
- grant_storage: ara::iam::internal::GrantStorage* {readOnly}
- result: ara::iam::ifc::grantqueryclient::GrantQueryResult = ara::iam::ifc::...
+ GetResult(): ara::iam::ifc::grantqueryclient::GrantQueryResult {query}
+ GrantCheckHelper(grant_storage_: ara::iam::internal::GrantStorage*, buf: char*, buf_length: std::size_t, logger_: ara::log::Logger&)
«use»
«use»
AccessManager
# data_: std::shared_ptr<AccessManagerData> {readOnly}
+ AccessManager(queryServer: std::unique_ptr<ara::iam::ifc::ipc::IPCServerAdapterInterface>, grantStorage: std::unique_ptr<ara::iam::internal::GrantStorage>, execStateClient: std::unique_ptr<ara::exec::ExecutionClient>)
+ ~AccessManager()
+ EndEventLoop(): void
# HandleNewClientConnection(clientSocketDescriptor: int, data: std::shared_ptr<AccessManagerData>): void
+ InitGrantStorage(grantStorage: ara::iam::internal::GrantStorage&, basePath: ara::core::StringView, logger: ara::log::Logger* const): bool
+ InitServerAdapter(server: ara::iam::ifc::ipc::IPCServerAdapterInterface&, logger: ara::log::Logger* const): bool
# ProcessPolicyQueryRequests(): void
# ReportStateRunning(): void
# ReportStateTerminating(): void
+ RunEventLoop(): void
+ SignalHandler(sigNum: int): void

Figure 2.5: Class diagram: AccessManager

The grants are stored in a single json file and are parse by the class JSONParser) If the file does not exists it is assumed that no grants are allowed. All allowed grants are stored in the class GrantStorage. This uses a std::map to efficiently access the stored grants. For the comparison the helper struct KeyWrapperLess is used.

To check if a binary serialization of a grant does exists in an instance of GrantStorage the helper class GrantCheckHelper is used.

Finally the AccessManager needs an instance of the ExecutionClient to indicated that all preprocessing (e.g. initial loading of all allowed grants) has been performed and it is now ready to process requests.

2.3 Dynamic Behavior

2.3.1 GrantQueryClient - API usage

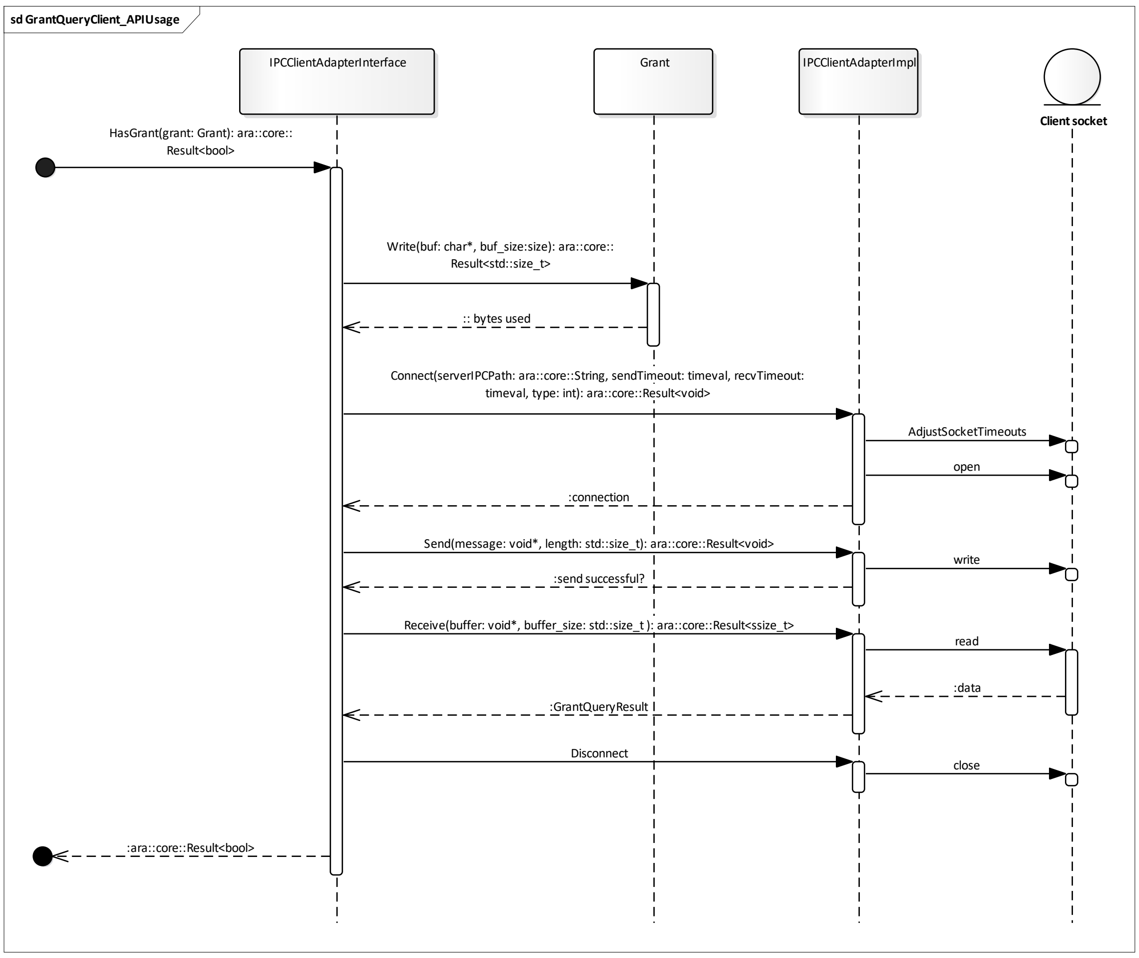
sd GrantQueryClient_APIUsage
IPCClientAdapterInterface
Grant
IPCClientAdapterImpl
Client socket
HasGrant(grant: Grant): ara::core::Result<bool>
Write(buf: char*, buf_size:size): ara::core::Result<std::size_t>
:: bytes used
Connect(serverIPCPath: ara::core::String, sendTimeout: timeval, recvTimeout: timeval, type: int): ara::core::Result<void>
AdjustSocketTimeouts
open
:connection
Send(message: void*, length: std::size_t): ara::core::Result<void>
write
:send successful?
Receive(buffer: void*, buffer_size: std::size_t): ara::core::Result<ssize_t>
read
:data
:GrantQueryResult
Disconnect
close
:ara::core::Result<bool>

Figure 2.6: Sequence diagram: GrantQueryClient API usage

To determine whether a user has access to a certain resource, a corrensponding grant has to be created and need to be passed to the HasGrant() method. If the adaptive application which is the initial trigger of the grant check needs to be identified, it is up to the caller or of the HasGrant() implement the necessary code. One possible way of doing such an identification is to optain the pid of the calling process using SO_PEERCRED (see GetPeerPID for a possible implementation) and use FindProcess() method of the execution manager. The caller has to ensure that the original calling process is still running and the pid has has not been reassigned during while the internal functionality to map a pid to a an adaptive application has been performed.

The HasGrant() is used to check if a grant is allowed. The HasGrant method is a wrapper to trigger the actual processing in a separate thread. In the worker thread an instance of IAMGrantQueryClientImpl will be created which will do the actual processing. In the diagram shown above only the part HasGrant call of IAMGrantQueryClient Impl is shown.

The IAMGrantQueryClientImpl will itself create an instance of IPCClientAdapterImpl to open an ipc connection to the AccessManager. The parameters for the ipc connection are specified in the ipc_parameters.h file. Each call of the HasGrant() method will trigger the creation of a new ipc connection. Connections will not be reused.

The grant to check needs to be serialized and transmitted over the ipc connection. The IAMGrantQueryClientImpl itself will wait for a return transmission from the AccessManager if the grants is allowed or not allowed or an error occurred during the processing of the request.

If an error or timeout with the ipc connection occurs or there is any other problem regarding the query an appropriate error GrantQueryClientErrc is returned.

Furthermore it is advised to disable the signal handling of SIG_PIPE. The GrantQuery Client is using pipes and if the communication with the access manager is not possible a SIG_PIPE may be triggered. If the signal handling of SIG_PIPE has not been updated, this may lead to a termination of the process using the GrantQueryClient.

2.3.2 AccessManager - Startup

The initialization of the AccessManager requires that an ipc server socket needs to be opened and the all allowed grants are loaded.

For the ipc socket initialization some parameters are needed. To simplify testing an already initialized instance of a class implementing the IPCServerAdapterInterface interface needs to be passed. This is usually an instance of IPCServerAdapterImpl, which provides socket ipc functionality to the AccessManager. The initialization may be performed by AccessManager::InitServerAdapter which passes the default parameters which are used by the AccessManager and the client library. The parameters for the ipc connection are specified in the ipc_parameters.h file.

The AccessManager also is initialized with an instance of the GrantStorage class, which stores the grant information. The given instance of the GrantStorage need to be initialized using the InitGrantStorage functionality, which parses the grant information based on the given configuration. For testing purposes, it also possible to manually add grants in the GrantStorage instance before passing it to the AccessManager.

Grants are parsed from the file /etc/system/iam/access_control_lists.json. The manifest file is processed by the JSONParser. Parsing of the file also includes a checking if the corresponding json schema returned by GetJsonSchema(). Grants are parsed and added to the storage container GrantStorage.

After the initialization phase is completed the access manger is notifying the execution manager that it is running and waits for incoming connections.

2.3.3 AccessManager - Handling of requests

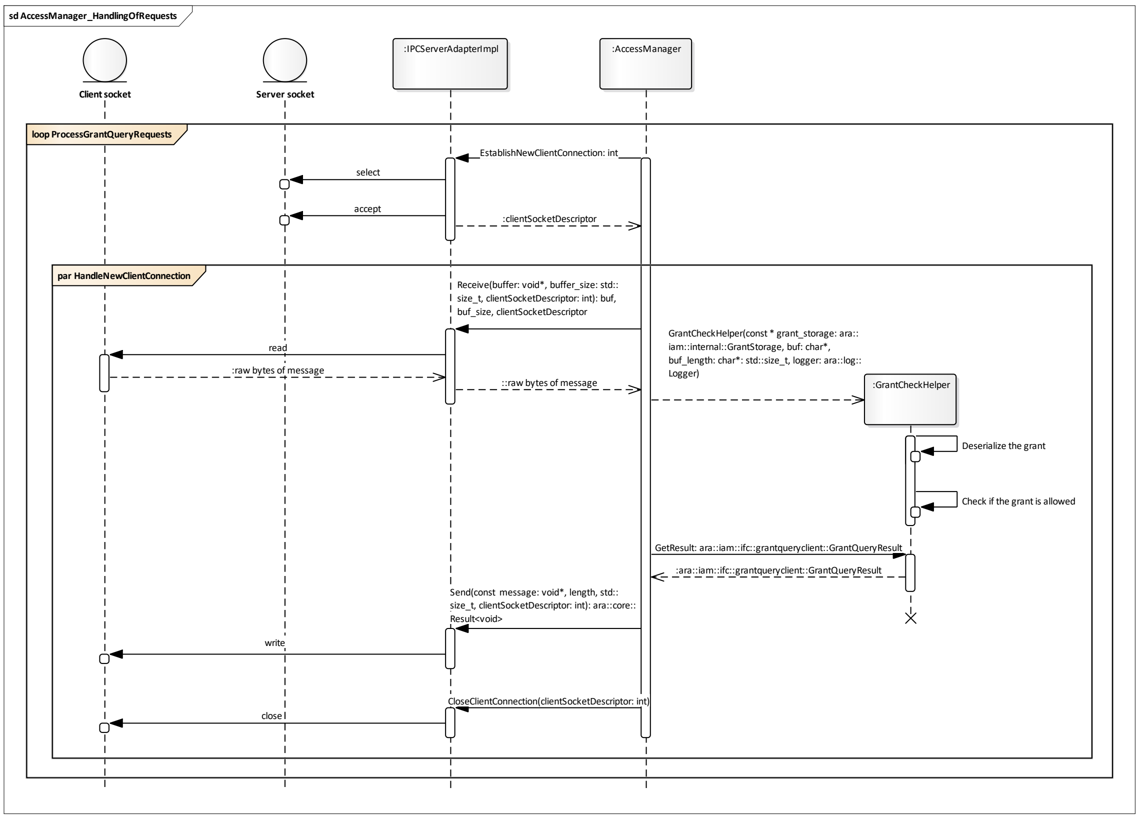
sd AccessManager_HandlingOfRequests
Client socket
Server socket
:IPCServerAdapterImpl
:AccessManager
loop ProcessGrantQueryRequests
EstablishNewClientConnection: int
select
accept
:clientSocketDescriptor
par HandleNewClientConnection
Receive(buffer: void*, buffer_size: std::size_t, clientSocketDescriptor: int): buf, buf_size, clientSocketDescriptor
read
:raw bytes of message
::raw bytes of message
GrantCheckHelper(const * grant_storage: ara::iam::internal::GrantStorage, buf: char*, buf_length: char*: std::size_t, logger: ara::log::Logger)
:GrantCheckHelper
Deserialize the grant
Check if the grant is allowed
GetResult: ara::iam::ifc::grantqueryclient::GrantQueryResult
:ara::iam::ifc::grantqueryclient::GrantQueryResult
Send(const message: void*, length, std::size_t, clientSocketDescriptor: int): ara::core::Result<void>
write
CloseClientConnection(clientSocketDescriptor: int)
close

Figure 2.7: Sequence diagram: AccessManager request handling

The loop uses the EstablishNewClientConnection method of the IPCServerAdapter Impl to listen for new connection requests on the socket. Even without a new connection this method will return after a predefined timeout, allowing for a periodic polling of the termination condition.

If a new ipc connection has been established, a new thread is spawned for it and the new request is processed within the HandleNewClientConnection method. The helper class GrantCheckHelper is used for the further processing. Before checking if the received grant is allowed the binary serialization of the received grant needs to be parsed. The list of allowed grants is kept in an instance of GrantStorage.

The result is sent back using the same ipc connection. If an error occurs during the checking of an appropriate error is returned.

2.4 Technical details

2.4.1 Storage of grant information

Currently there is only one type of grants GrantDemoFc. Thefore the possible file structure of the allowed grants is rather limited.

Grants are stored in a json manifest file which is located in the file /etc/system/iam/access_control_lists.json.

The grants follow a whitelist approach. If no file containing any grants has been found a warning message is logged, that IAM will always return that any requested action will be forbidden.

2.4.1.1 Example

Excerpt from a manifest file, showing the allowed grants:

```
(
    {
    "grants": [
        {
            "type": "DemoFCGrant",
            "Process": "Processes1",
            "DemoFile": "DemoFile42"
        },
        {
            "type": "DemoFCGrant",
            "Process": "Processes8",
            "DemoFile": "DemoFile42"
        }
    ]}
)
```

2.5 Usage of the GrantQueryClient API

2.5.1 Build system setup and installation

To use Grant Policy Query API in a CMake-based application, the following CMake command must be issued:

To use the grant query client you need to add ara-iam as an additional dependency to your yocto receipt:

```
DEPENDS += " \
    ara-iam \
"
```

In your CMake project you need to add the dependency to ara-iam and Poco as well and a a dependency to your target (e.g. ${PROJECT_NAME}):

```
find_package(ara-iam REQUIRED)
find_package(Poco REQUIRED Foundation)
# [...]
target_link_libraries (${PROJECT_NAME} ara::iam)
```

2.5.2 Code

The code to check if a grant is allowed consists of one single call to the grantqueryclient api and checking the return value.

Please note that the grant may require additional information to identify the adaptive application, which is requesting access. The identification of the adaptive application needs to be performed by the caller of the grant query client. It is vendor specific since different implementations or possible. In this example the identification of the calling application is not shown.

```
#include <iostream>

// Include for the grant query client itself
#include "ara/iam/ifc/grantqueryclient/grant_query_client.h"

// For the example the GrantDemoFC is checked. If another grant
// is checked its appropriate header files needs to be included
#include "ara/iam/ifc/grant/grant_demo_fc.h"

int main()
{
    // Create the grant you want to check
    ara::iam::ifc::grant::GrantDemoFC grant(ara::core::InstanceSpecifier("
   test"), ara::core::InstanceSpecifier("test"));

    // Check via the PDP if the grant is allowed
    const ara::core::Future<bool> resFuture = ara::iam::ifc::
   grantqueryclient::HasGrant(grant);

    // Wait for the Future to obtain the result
    const ara::core::Result<bool> res = resFuture.GetResult();

    // Check the result
    if (res.HasValue())
    {
        if (res.Value())
        {
            std::cout << "grant is allowed" << std::endl:
        }
        else
        {
            std::cout << "grant is forbidden" << std::endl:
        }
    }
    else
    {
        std::cout << "some kind of error occurred" << std::endl:
    }
}
```

2.6 Used OSS components

This sections lists all OSS components used in the implementation of the Functional Cluster, their architectural context and the rationale for their use.

OSS	Version	Explanation	Binding	License	License Version
GTest/GMock	1.7.0	SDE (Unit Testing)	static library	BSD-3-Clause	NA
RapidJSON	1.1.0	JSON Functionality	header only, static linked	MIT	NA
BOOST	1.68.0	Usage of boost::variant	static (header file inclusion)	Boost Software License	BSL-1.0
POCO C++ Libraries	1.9.0	Usage of serialization	dynamic library	Boost Software License	BSL-1.0