INTERNATIONAL STANDARD

ISO/IEC 4396-3

First edition
2023-12

Telecommunications and information exchange between systems — Recursive inter-network architecture —

Part 3: Common distributed application protocol

Télécommunications et échange d'information entre systèmes — Architecture récursive inter-réseaux —

Partie 3: Protocole pour les applications distribuées CDAP

Reference number
ISO/IEC 4396-3:2023(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 6 *Telecommunications and information exchange between systems*.

A list of all parts in the ISO/IEC 4396 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The common distributed application protocol (CDAP) is used by communicating recursive inter-network architecture (RINA) applications to exchange application-specific data and effect remote actions. CDAP implements the protocol messages and state machines to allow the two endpoints of a communication flow to exchange read/write, start/stop, and create/delete method invocations on remote "objects". The semantics of those objects and operations are opaque to the CDAP protocol itself. Because CDAP is not specific to RINA, it can be used by any distributed application that needs to share information or initiate state changes with another application over a network. CDAP is used in the RINA architecture by applications, and specifically by inter-process communication (IPC) Processes, which are specialized applications that cooperate to create a Distributed IPC Facility (DIF) that provides network transport to other applications.

Telecommunications and information exchange between systems — Recursive inter-network architecture —

Part 3: Common distributed application protocol

1 Scope

This document provides the common distributed application protocol (CDAP) specification. CDAP enables distributed applications to deal with communications at an object level, rather than forcing applications to explicitly deal with serialization and input/output operations. CDAP provides the application protocol component of a distributed application facility (DAF). CDAP provides a straightforward and unifying approach to sharing data over a network without having to create specialized protocols.

This document provides:

— an overview of CDAP;

— the specification of CDAP;

— a description of policies, in the specific sense introduced in the text;

— notes on the context of CDAP.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 4396-1, *Telecommunications and information exchange between systems – Recursive Inter-Network Architecture — Part 1: RINA Reference Model*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 4396-1, and the following apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1
application object model
common distributed application protocol (CDAP) object model used in the application connection (AC) data transfer phase.

Note 1 to entry: If an AE is capable of supporting multiple object models, this selects the one to be used for this AC. If not, the requested object model value should match that of the sole implemented model.

3.2
CDAP message encoding rules
encoding of bits sent for all CDAP messages in an application connection

Note 1 to entry: The concrete syntax to be used for the AC is determined by the Common Application Connection Establishment Phase (CACEP) and provided to the CDAP data phase in the application connection state vector (ACSV).

3.3
CDAP policy
CDAP implementation option where something can be done in multiple ways, and no single choice is mandated

Note 1 to entry: CDAP implementers should make a choice of how to implement every CDAP policy; recommended policies are provided in this document.

3.4
CDAP profile
specific and complete definition of all the policies in CDAP and the object model that should be implemented as described in order to create a conforming, compatible implementation

3.5
CDAP syntax version
abstract syntax
integer that indicates version of abstract syntax used

3.6
method call
request for the recipient application-entity (AE) to make a call to a method (function) specific to an object

3.7
object
named conceptual data entity residing within the resource information base (RIB) view made available to an application connection (AC) by the application

Note 1 to entry: Objects have a class (type), a name and possibly an interchangeable ObjectID (OID), and a value. The object [in an unspecified way, as object/class models and inheritance are unspecified by common distributed application protocol (CDAP)] provides a set of methods (functions) that correspond to the CDAP message types, which can become the targets of method calls on that object during the data transfer phase of the AC.

3.8
object method
method (function) associated with an object that will be invoked by receipt of a corresponding CDAP message

3.9
object model
entire set of objects available to an application connection (AC) and their collective relationships and behavioursNote 1 to entry: The object model to be used for an AC is selected by the Application Object Model variable set during the Common Application Connection Establishment Phase (CACEP).

3.10
object model version
specific named object model used for an application connection

Note 1 to entry: Common distributed application protocol (CDAP) does not dictate the form of Object Model Version names/values; values are mutually agreed-upon by a priori application agreement, determined by Common Application Connection Establishment Phase (CACEP), and possibly made available in the application connection state vector (ACSV) for use by objects.

3.11
policy point
place to optimize or configure a common distributed application protocol (CDAP) implementation to perform some operation

Note 1 to entry: Policy points have the form POL-CDAP-x.

4 Description of CDAP

4.1 CDAP –RINA application protocol

The word “application” is used in this document in a general sense. Independent application programs, individual instances of a distributed program, or even independent threads of a single process execution can be considered to be “applications” if they communicate via CDAP. In typical RINA usage, the “applications” are Application Entities (AEs) residing within two instances of a Distributed Application Process connected by a RINA flow.

The objects that CDAP operates upon are mutually-agreed-upon representations of the exposed data and entities between two communicating applications. CDAP does not assume that the applications share any directly-accessible memory, so CDAP is an example of a Remote Procedure Call protocol. RINA refers to the set of these known objects as a Resource Information Base (RIB), a virtual database of objects that the two applications are mutually aware of and able to address via CDAP.

CDAP does not assume that the RIB exposes the entire state of either application, or that the objects manipulated by CDAP map 1:1 onto actual state variables or entities; CDAP simply communicates requested operations on objects that the communicating applications choose to expose. The objects can actually be virtual objects synthesized for external presentation or to generate side effects; they can be considered to be “projections” or “views” of implementation objects, created for convenience or abstraction purposes. While a hierarchical tree-based organization is often used to unambiguously name and locate objects in a RIB, CDAP itself imposes no particular structure.

CDAP borrows from Common Management Information Protocol (CMIP) a straightforward standards-defined protocol and shares many properties with other remote-method-invocation based protocols such as the Bell Labs Plan 9 “file protocol”. While known concepts are employed to benefit from the experience of prior implementations, the design of CDAP explicitly takes into account that implementations will be done in multiple languages and on various platforms of widely different scales, so some lowest-common-denominator choices have been adopted, and many details have been left undefined in the base protocol to allow tailoring CDAP implementations for their end use.

CDAP does not define a single implementation specification, but is rather a framework in which specific application, communication, and scale requirements can be met while still providing higher-level application code with a consistent set of operations and behaviors. CDAP does this by providing a core set of mechanisms with a variable set of policies that allow but encapsulate and limit the desired variability. Specific instances of the set of variable aspects of the protocol, e.g. message encoding syntax, that are characterized in this specification as policies, can be standardized to ensure interoperability of different implementations. This document provides default policies as examples; in the absence of problems with these defaults for a specific usage, it is recommended that CDAP implementations adopt them to promote reuse and prevent unnecessary divergence.

4.2 Application entities (AEs) within applications

RINA defines an AE as a portion of an application that performs some distinct function. An AE communicates with one or more compatible AEs in other applications to accomplish a specific function of the application. For example, one AE in an application can deal with database access, another with authentication, and another with reporting and management, each of which operates over and needs access to a specific, and potentially distinct, set of remote data that is relevant to its function.

Within each communicating application, there will be one or more AE that performs a particular purpose of communication, with a corresponding set of objects that have state, a usage protocol (i.e. legal sequences of operations on the objects), and specific semantics. In practice, AEs may be implemented via sub-routine libraries, threads, objects, or some other processing model, but the AE concept is present whether explicit or implicit.

Application connections are created between an AE instance of one application and a corresponding, compatible AE instance of another application. Applications may implement any number of AEs, and those may conduct any number of application connections, with any number of other applications.

From a security standpoint, it is important to restrict the access permissions of an AE to only those RIB objects relevant to its operation. While CDAP does not itself implement that separation, it provides applications with the information needed to do so.

4.3 Objects

All modern programming languages can implement the concept of an object that encapsulates data and a set of operations upon it. Properties, e.g. fundamental types, inheritance, extensibility, safety properties, aggregation capabilities, are richer in some languages than others. CDAP implements a bare-minimum model that can interface with virtually any existing object-oriented implementation approach. In the CDAP model, the AEs that are communicating with one another via an application connection share an object model that has a defined set of objects, object classes, and naming structure for the objects. Each object implements a specific set of method (function) calls that operate on the object.

4.4 Method calls on objects

CDAP messages each encode a method call, with arguments, that represents a request for a remote AE to perform that method call on one of its objects. A CDAP protocol implementation provides the mechanisms for a sequence of steps that include encoding the desired method call and its arguments into a message, sending the message over the RINA flow, decoding it, presenting the method call to the addressed object, possibly returning a result message, and completing the original method call. A goal of CDAP is to allow doing this sequence of steps in a common way, enabling a standardized programming API, in the face of different policies for, e.g. encoding syntax, implementation language, object models.

4.5 Object model

CDAP encodes read/write/start/stop/create/delete method calls on a set of objects that is specific to an application connection. A meaningful CDAP exchange requires that the applications agree fully upon the set of objects that can be addressed and their exposed properties, operations, and semantics. In this document, the "object model" is used to refer to the entire definition of the set of objects, including their class, their data description, their names and naming conventions, their interrelationships, their methods, their usage pattern, and their behaviors such as side-effects. An application connection is associated with a specific object model, determined when the application connection is established. An application can implement and expose, through its AEs, as many object models as appropriate to its purpose.

The underlying application data exposed in an object model as an “object” can actually be organized differently than it is presented to the other AE in the object model. These “virtual” objects may represent application data that needs to be aggregated, selected, computed, or iterated over in order to satisfy the semantics of a requested CDAP operation on the target virtual object. Any such operation is performed by the methods of the (virtual) objects that are exposed, not by CDAP.

Since objects and their semantics generally evolve with time, the Common Application Connection Establishment Phase (CACEP) protocol that is used to establish an application connection allows specifying at connection establishment time a “version” of the object model that will be used for the duration of that application connection. This makes it possible for an application to discover a potential incompatibility between the object model being used by the other application, and/or to have a vocabulary of multiple object model versions to use, and to choose the correct one to use for a particular

application connection depending on its partner application. This is especially useful during transition periods when an object model is being revised, and updated versions of applications that understand the new model are not yet fully propagated.

There is no CDAP-defined object model or set of objects that all applications using CDAP should implement. Object models are not part of CDAP itself.

4.6 Application connection

Application cooperation requires the applications to agree upon the syntax of the CDAP messages (CDAP can be encoded in multiple ways) and the object model, and further, to agree upon the privileges that each application grants to the other with respect to its own objects, based on identity, capabilities, or other security mechanism. An application connection operates within an establishment process, the CACEP, that authenticates the applications, determines the message syntax and object model to use, and provides these parameters for use in an application connection, prior to the data transfer phase where CDAP messages are exchanged, and operations are performed on objects.

4.7 Application connection state vector (ACSV)

The CACEP, either by message exchange, defaults, or other mutually-agreed-upon method (the specifics are policies of the applications and of CACEP, and are completely opaque to CDAP), determines the values of key application connection parameters before handing off operation on the flow to the data transfer phase. This information is provided to the CDAP implementation via an Application Connection State Vector (ACSV), which CDAP will consult when needed as it interprets messages. While the ACSV is not exchanged via CDAP messages and is therefore not specifically defined by the CDAP protocol, the contents are relevant to some policies, especially CACEP, which should provide some of the contents of the ACSV, and to some objects, e.g. those that should perform access permission or object model checks. Therefore, the ACSV is defined as a visible CDAP concept to ensure that the requirements that it places on CACEP and CDAP policies have a place to be made explicit.

4.8 Requestor and responder roles

Many interactions between two applications are asymmetric, such as a classic client/server exchange, but CDAP builds in no such constraint. The CDAP protocol is symmetric, in the sense that either side may make method calls via CDAP messages on objects of the other side, without regard to which side initiated the application connection. In this document, when the initiator of a message if referred to as the “requestor”, and the recipient of the message that (optionally) generates a reply as the “responder”, it is in the context of a specific message exchange, not the original flow or application connection establishment. The application AEs can have a specific role in the application connection and behave asymmetrically, but it is not relevant to CDAP operations.

4.9 Validation of values/operations by CDAP

The CDAP protocol provides the mechanism for communicating method calls with values, but has no inherent understanding of the semantics of objects or the meaning of the object fields and values. Those remain the responsibility of the applications and their object models and the policies they use. However, CDAP messages themselves have encoding and consistency requirements. Messages are parsed, generated, and have common operations performed on them (such as tracking replies) using mechanisms common to any object model, and many errors in forming or parsing CDAP messages are detected by those mechanisms of CDAP itself. Other semantic correctness requirements on the operations carried by CDAP messages, e.g. the range of a field value in a READ or WRITE message, are enforced by object methods; in such cases, CDAP mechanisms (such as sending a log message or even abandoning the application connection) may be invoked by objects to request CDAP mechanism assistance to respond to these errors.

To enhance the ability of objects to respond to and validate application connection specific parameters such as authentication, CDAP passes application connection parameters provided in the ACSV to object method operations for their examination. It is the responsibility of the method implementations

to validate the parameters extracted from a message, to restrict access to objects as necessary, and to generate and present appropriate views of objects, in accordance to the application connection parameters in the ACSV. For example, in many applications CACEP will have established credentials that may then be used to validate access permissions (CDAP's only role in this is to provide the credentials, via the ACSV, to objects for use in that access validation). In practice, such common actions should be factored out and performed by mechanisms outside the actual object method implementations. This is especially important for security-related operations such as access permission validation, to avoid introducing a defect opportunity into every object that is expected to perform a security check.

4.10 CDAP application programming interface (API)

This document does not specify an API for generating requests or receiving replies during the CDAP data phase. It only defines the messages that are created and consumed by such interfaces. The CDAP programming API is an application implementation choice. In fact, CDAP and object operations can be implicit in data operations on a RIB, with no explicit CDAP API provided. However, since all CDAP implementations have a common set of mechanisms, a common API that operates across multiple CDAP variations is straightforward to create.

4.11 Standardization and policies

The purpose of standardization of a protocol is to enable interoperability. This is achieved by intentionally limiting flexibility, typically by choosing a specific, usually single, way that implementations should encode communications, define the semantics of communication exchanges, and define all aspects and behaviors of the objects being communicated about. When a protocol is intended for a specific application, freezing options and making optimized trade-off choices is both possible and necessary.

However, the CDAP protocol is intended to apply to a broad range of applications, so rather than narrowing it to a specific range by setting all details in concrete a priori, this specification provides a framework that allows defining a variety of optimized fit-for-purpose CDAP protocol implementations.

In many parts of this document an implementation choice is designated as being a policy and a name for the policy is provided to aid in defining and documenting specific CDAP implementations. While there is flexibility provided in message encoding and other aspects of CDAP itself, the vast majority of variability and policy decisions involve the object model (the specific objects being manipulated via CDAP and their behaviors), which by their nature are application-specific. When a complete set of policies has been specified for a specific CDAP usage, including the object model, that specification combined with this specification defines a CDAP Profile. Two AEs that implement the same CDAP Profile can communicate meaningfully. Two AEs that implement different CDAP Profiles, in general, cannot.

In order to reduce unnecessary divergence among implementations, many suggested default policy choices are provided. In the absence of a compelling reason (e.g. highly-constrained resources or a pre-existing standard that constrains some aspect of the application), these default policy choices are recommended to be adopted. This topic is discussed further in 5.1.

5 Specification

5.1 CDAP profile — Policies and standardization

CDAP provides applications with a fixed set of methods, corresponding 1:1 to CDAP message types, to effect operations on remote objects via exchange of messages. In this document common properties of all CDAP implementations such as these are referred to as the mechanisms of the protocol. The mechanisms for exchange of CDAP operations are defined in this document, but the bit-level encoding of messages and the naming, values, types, and semantics and behaviors of the objects do not have a mandated single definition. These designed-in variability points are identified in this subclause as named policies that may vary, and are specified in detail in that context, when CDAP is used for a specific application. Policies can be selected to optimize a CDAP implementation for (e.g. size, speed, readability, flexibility, reuse, compatibility), to best serve the application needs.

When defining an interoperability standard for applications that use CDAP, a CDAP Profile is defined and mandated as part of that standard. For every named CDAP policy point described in this document, a CDAP Profile specifies the policy that compliant applications will use. To encourage commonality across implementations, a suggested default policy is provided for each policy point; object model definitions should adopt the default policy unless there is a compelling reason not to do so.

5.2 Application connection establishment

Application connection establishment in the RINA architecture is performed by the CACEP as described in which is loosely patterned after the Association Control Service Element (ACSE) protocol. The details of how the CACEP phase performs its function are not relevant to CDAP itself. To operate properly, CDAP depends upon establishment of the necessary set of initial conditions and parameter values prior to commencing CDAP operations. Conceptually, and commonly in an implementation, those values are communicated using the ACSV described below.

The data transfer phase is entered after the CACE phase completes successfully. The application connection ends when either application closes the flow, or detects that the flow has closed, or the application informs CACEP that the application connection is complete. All application connection information is discarded at that time, and a CACEP phase should be performed again if it is desired to resume operation.

The following is a summary of the CACEP process to show how CDAP fits within the enclosing establishment framework.

Steps for creation, parameter setting, and destruction of an application connection:

a) Establishment of a data flow (communication channel, or colloquially “connection”) between the applications.

   For full functionality, CDAP requires that the flow delivers Service Data Units (SDUs) transparently, in-order, loss-free, and with negligible error rate. Partial functionality can still be provided if the in-order and loss-free requirements are not met. The RINA flow allocation process with appropriate Quality of Service (QoS) parameters can be used to establish such a flow.

b) Establishment of parameters for an application connection.

   This phase first identifies the CACEP syntax and then carries out the CACEP protocol exchange, which is implemented via customizable CACEP policy, that authenticates the applications to each other to their mutual satisfaction (the policy can range from “no authentication required” to a bi-directional mandatory authentication). It also determines and initializes, again via CACEP policy, key parameters for the CDAP data phase, saving all required information in an ACSV for use by the AC and by RIB object methods. Fundamentally, CACEP establishes and then hands off a fully initialized application connection description to CDAP.

c) The application connection data transfer phase.

   If the CACEP process proceeded to successful completion, the preconditions for CDAP execution will have been established, and AC parameters will have been stored in the ACSV. The applications then enter the data transfer phase of the AC in which CDAP messages are exchanged. The ACSV provides all necessary information for further CDAP and object operations.

d) If the CDAP AC is no longer needed, either application typically ends the AC by requesting CACEP to terminate it. CACEP would typically destroy the flow, but if the flow is reused, care should be exercised to ensure that no residual information from a previous AC persists except the flow and its properties.

5.3 Application connection state vector (ACSV)

The ACSV is provided to CDAP by the CACEP at the completion of the CACE phase, which begins the data transfer phase by invoking CDAP initialization logic with the ACSV. The CDAP initialization logic

will examine the ACSV and may choose to reject the AC and close it immediately if it is incomplete or inconsistent. It is recommended that the CDAP implementation provide CACEP with enough information to preclude initiation of an AC that cannot be supported by CDAP, or to participate with CACEP in negotiating the parameters. The precise contents of the ACSV are a policy of a specific CDAP Profile [POL-CDAP-ACSVcontents]. Contents of the ACSV may include implementation-specific information, but shall include at least the following fields that are referenced elsewhere in this document:

a) CDAP message encoding concrete syntax specification [type integer, value (POL-CDAP-CSYNTAX)].

CDAP messages can be encoded using any desired syntax, provided that the implementations can appropriately encode and decode it. The only requirement is that both applications can encode and decode CDAP messages using the selected syntax correctly and that the syntax can encode the full range of values needed for CDAP message fields and for each of the addressable objects.

b) CDAP syntax version [type integer, value (POL-CDAP-OBJ-VERSION)].

This is a value specifying which version of the CDAP specification is in use. It is solely intended to allow the CDAP message encoding (fields and their meanings) to evolve if necessary, and is intended to change only if incompatible changes should be made to the fundamental CDAP message types or fields. An application shall reject an AC if the Syntax Version does not correspond to a CDAP message syntax that it can accept and generate.

c) The flow (type flow_handle).

This is used to perform I/O operations to/from the flow. Implementations may allow the handle to be queried to determine, for example, the QoS parameters or other properties of the flow. For example, if the QoS of the underlying flow indicates that it is unreliable, the application cannot assume that messages will always be received, or if received that the sender will always receive a requested reply, so the application should limit its CDAP operations to a subset that can withstand such uncertainty or should terminate the AC. The specific operations available on the flow_handle are implementation-dependent and not specified by CDAP, but a specific capability may be implied by specific policies.

d) The AE name requesting the AC (type string).

This may be used by the application to further restrict or select the set of objects in the CDAP object model that can be operated upon by this AC [POL-CDAP-OBJ-VISIBILITY]. The meaning of this parameter is opaque to CDAP (CDAP's only role is to make it available to CDAP and to method calls invoked via CDAP messages).

e) The Object Model to be used for the AC (type integer, value described by [POL-CDAP-OBJ-VERSION]).

In the event that multiple versions of the object model used by the AEs of the communicating applications exist, e.g. due to evolution of the object model, incomplete rollout of updated implementations, or unintentional implementation discrepancies, this value specifies which one the AC will use. If an AE implements a single object model, this value should match the value representing that model or the AC shall be terminated. This value is opaque to CDAP, it is made available to the method calls invoked by CDAP messages, which may adapt their behaviour to correspond to the version in use for the AC, but it is not examined by CDAP mechanisms.

f) Security information.

If CACEP establishes an identity or other credentials to be used in validating permission [POL-CDAP-AUTH], this field provides privileges/identity/credentials of the other application for purposes of limiting the AC's access to objects in the view. The field includes the mandatory Verbose. The remainder of the field, if any, is specified in [POL-CDAP-AUTH]. The Boolean value Verbose in this field indicates to CDAP mechanisms whether CDAP encode/decode and other internal CDAP operations are to return detailed information on success or errors (if the Boolean is TRUE), or simple SUCCESS/FAILURE return values (if the Boolean is FALSE), allowing the policy to control potential leakage of implementation information to less-trusted applications by CDAP mechanisms. The Boolean may also be referenced for the same purpose by object methods

when they create return values. CDAP's only other role in security policy is to carry the CACEP-established authentication information, if any, to the object method operations invoked via CDAP.

Additional implementation-specific information about the flow or AC, e.g. application names and instance ids or which application initiated the flow, which do not appear in CDAP proper, can be included in the ACSV by implementations. Such information is not used in this reference and is therefore not mandated by CDAP, but can be useful to the application or to specific objects and can be very useful for debugging, logging, or future enhancements.

5.4 Objects and the object model

5.4.1 Object properties

The RIB objects that are visible to an AC have five primary properties of concern to CDAP:

a) The class the object belongs to. This property recursively captures the type(s) of the data field(s) of the object and those of their contained data fields (the word "type" is sometimes used, especially when the methods are not included in the discussion, and generally the word "class" is used when including the methods in the discussion).

b) The set of methods that are provided to operate upon the object.

c) A path name that is unique among the objects available to the AC.

d) A value for each of the data fields of the object.

e) An object identifier, or ObjID, which is an integer assigned to a specific object as a synonym to its name; like the name, it shall also be unique among all objects available to the AC.

5.4.2 Object model definition

The object model in a specific CDAP Profile is defined by policy [POL-CDAP-OBJECTMODEL]. Within the object model policy are sub-policies:

— The version (values are application specific) of the object model to use for the current AC [POL-CDAP-OBJ-VERSION].

— The visibility rules for objects based on AE, authentication, or other factors [POL-CDAP-OBJ-VISIBILITY].

— The rules for object names shall be specified explicitly (e.g. hierarchical vs. flat, syntax, separators, alphabet, …), [POL-CDAP-OBJ-NAMING].

— The rules for objID values shall be specified explicitly (e.g. pre-assigned values, rules for generating new objIDs, …), [POL-CDAP-OBJ-objID].

— The use of objIDs vs. object names to identify objects shall be made explicit, [POL-CDAP-OBJ-ObjRef].

— Every object class shall be fully defined, including names, type, and value-consistency constraints on all exposed fields; fields may be designated as "private" to permit specifying method behavior – this is for specification purposes only, such fields are never made visible via CDAP, [POL-CDAP-OBJ-CLASSES].

— For each object, or group of objects with identical behavior, the methods implemented by the object and any pre-conditions that shall be satisfied before invocation of each shall be specified, as well as the meaning of any return values from the methods; all objects in a group shall have the same class but not all objects of a given class shall implement the same methods, [POL-CDAP-OBJ-METHODS].

— For each object or object group (e.g., Object Class), any specific authentication/access restrictions on each method and field of the object, as required by the authorization policy as specified in [POL-CDAP-AUTH].

— As part of the CREATE method description for each object, if implemented, rules for creating new objects, including any restrictions on their names/objIDs, shall be stated explicitly, with any additional global requirements in [POL-CDAP-OBJ-OBJCREATE].

— All initial mutually-agreed-upon a priori objects and structure, if any, shall be fully defined in accordance with other policies and given an initial value. [POL-CDAP-OBJ-INITIAL].

In a given model, many of the properties and requirements of individual objects will be inherited from general rules in the object model and/or from the class of the object, rather than repetitively defined individually. For example, the methods for each object can be inherited from its object class, and access permissions on objects can be specified by a policy that applies to all objects.

Objects may represent actual data in the RIB of an application, or may simply be emulated on-demand by method calls on virtual objects. To provide maximum flexibility to implementations, the objects should, when possible, be defined in such a way that remote references to the object should not be able to distinguish which is the case.

CDAP provides no built-in method to selectively access a data field or fields of an object. Therefore, such operations over an object model would in general be implemented either via virtual objects whose methods provide the needed operation, or via a policy such as naming. Any such convention shall be defined in the object model.

5.4.3 Object model version

The Object Model Version [POL-CDAP-OBJ-VERSION] to be used for an AC is determined during the CACE phase. The model is selected during CACEP, using a CACEP policy that may for example combine factors such as Application Name, AE Name, and Object Model value. If multiple Object Model Versions are supported in a CDAP Profile, every difference between object behaviors between the versions shall be explicitly documented and each difference associated with the version(s) that reflect it. Objects may modify their behaviour based on the Object Model Version in the ACSV in such cases.

5.4.4 Object class

Objects generally have a data portion (but need not, though not having a data portion eliminates the ability to perform READ or WRITE operations on the objects) that is recursively defined as being any of:

a) a scalar whose type is chosen from a defined set of built-in scalar types,

b) an aggregation of data items all with the same type (an array), where the type can itself be any defined scalar or aggregate type, or

c) an aggregation of data items having possibly-dissimilar types (a structure), where any field can be a scalar, array, or structure.

All objects also have an associated set of methods, which are functions specific to that object (possibly shared with other objects, often with all others of the same class), which perform specific operations on that object. The methods that are provided to operate upon the data can be inherited from the class or be specific to an object (the definition is provided in the Object Model). Built-in scalar types may have default string-valued names that can be used explicitly when creating objects with a scalar data type, as described by policy [POL-CDAP-OBJ-Types]. All other class names are string-valued, assigned by the object model. The class of an object is determined at its creation time and unchangeable during its lifetime; the class is determined by the object's CREATE CDAP message per the [POL-CDAP-OBJ-CREATE] policy.

5.4.5 Object name

Objects can be identified as the target of a CDAP message by specifying an ObjName, a string in the CDAP message that uniquely designates the object, or by using a synonymous unique integer, known as an Object ID. How names and Object IDs are used in messages to identify an object are dictated by

policy, and either or both may be used, as permitted by the policy [POL-CDAP-OBJ-ObjRef]. Names are specific to an object model, and a set of specific names and naming structure shall be agreed upon a priori by the application AEs prior to the AC in order to be able to meaningfully communicate operations on objects known to both parties.

While object names, encoded in messages as strings, frequently reflect an organization, such as a tree with "leaf" nodes representing objects and "directory" nodes serving to define the structure of the tree, the syntax and semantic conventions for naming are flexible, and are defined by CDAP policies [POL-CDAP-OBJ-NAMING, POL-CDAP-OBJECTMODEL].

5.4.6 Object ID — Shorthand name alias

The Object Identifier (Object ID or ObjID), encoded as an integer, is a short alternate name for an object. It can be used to shorten messages and avoid the necessity of repeatedly looking up objects using ObjName strings. Whether the name or the object identifier, or both, may be supplied in a message to identify an object is defined by a specific policy that describes how objects are to be referred to in messages [POL-CDAP-OBJ-ObjRef], and may in some cases be constrained by the capability of the message encoding syntax [POL-CDAP-CSVersion].

If ObjIDs are in use, an object may have an ObjID with a pre-arranged value known to both applications, or the value may be dynamically assigned for use in an AC. In the latter case, an ObjID value shall reference the same object for the duration of the AC, or for the lifetime of the object if shorter. Except for the reserved value zero, which can be used in messages to indicate that no ObjID is being provided, there is no CDAP-defined meaning to the numerical value or properties of the ObjID, though applications may choose to assign such meaning per their policy.

An ObjID may be sent to an application, e.g. in a reply message, by the owner of an object for use in future requests. Object model creators should consider carefully whether to permit the name of an object to be returned in a reply to a request that addressed the object solely with an ObjID, as this enables a class of enumeration attacks on a RIB, but this is a policy option and shall be explicitly specified in [POL-CDAP-ObjRef].

5.5 Messages and replies

CDAP messages encode remote method calls with arguments on a remote object using a specified concrete syntax. The canonical message exchange consists of sending a Request-type message, eventually followed by receiving a matching Reply-type message indicating the success or failure of the requested operation. Reply messages shall be requested explicitly in request messages. This allows, for example, for a reply to be foregone if the application would not change its behavior based on the result that would be returned in the reply, for example sending a sequence of operations whose final message can request a reply that reflects the success or failure of the entire sequence.

CDAP provides a mechanism to allow multiple request-reply exchanges to be in-progress at the same time, and for replies to be returned in or out of order ("split transactions"), by allowing a transaction identifying number called an InvokeID to be included in a request and subsequently used to associate a reply with its corresponding request. Use of the capability is optional. When using this capability, multiple requests can be sent without waiting for replies, allowing new operations to be initiated without waiting for opposite-end computation or network operations to complete. CDAP does not mandate replies to return in the same order that requests were sent, though object models and applications are free to restrict behavior to a blocking request-reply, or asynchronous, but in-order. Some applications may choose to process requests and generate replies in order as a simplification to program logic, while some may obtain a performance gain and therefore accept the higher complexity of out-of-order operation. To ensure interoperability, any limitations that restrict fully asynchronous operation with out-of-order replies shall be specified in the object model [POL-CDAP-OBJECTMODEL].

5.6 Message encoding

The encoding of the message type and other fields in the message into a form used for communication on a "wire" (flow) is referred to as the encoding rules of the message. Agreement between communicating

applications on the encoding rules to use is a fundamental requirement for communication. For CDAP, this is determined prior to the start of the CDAP data phase and provided to CDAP in the ACSV. This value will select from the set of encoding rules available to the implementation; if there is only one, then the value shall match the sole implemented rules for communication to take place.

As long as the encoding is capable of carrying the required range of argument types and values needed by the object model, any encoding can be used.

5.7 Methods on objects

An object method invocation is requested on a remote object by sending the corresponding CDAP message and arguments to the object in a Request-type message. Upon receipt, the message is parsed, the object is identified, the arguments decoded and defaults supplied where missing, and the object's method corresponding to the message is invoked. An attempt to make a method call on an object that does not implement the method results in a reply message with an error code, if a reply is requested, or the attempt is simply discarded if not (though it may generate an implementation-defined error response such as a log message). The method/message type is encoded in the Opcode field of the message, and additional values are encoded in other fields according to the message type, as specified below. Reply-type messages are only used in direct response to a corresponding Request-type message (i.e. same InvokeID) and use some of the same fields as well as additional ones to return the result from the method invocation.

Table 1 provides the entire set of messages and corresponding method invocations that are requested via CDAP messages. Each message also carries arguments to the method call; the description of the arguments and their types are also provided.

Table 1 — CDAP message types

Message name/Opcode	Purpose
CREATE	Create an object
CREATE_R	Response to A_CREATE, carries result of create request
DELETE	Delete a specified object
DELETE_R	Response to A_DELETE, carries result of deletion attempt
READ	Read the value of a specified application object
READ_R	Response to A_READ, returns result and (possibly-incomplete) value of the object
CANCELREAD	Cancel a prior A_READ request that has not completed
CANCELREAD_R	Response to A_CANCELREAD, indicates outcome of cancellation
WRITE	Write a specified value to a specified object
WRITE_R	Response to A_WRITE, carries result of write operation
START	Start the operation of a specified application object, typically used when the object has operational and non-operational states
START_R	Response to A_START, indicates the result of the operation
STOP	Stop the operation of a specified application object, typically used when the object has operational and non-operational states
STOP_R	Response to A_STOP, indicates the result of the operation

5.8 CDAP message

5.8.1 General

Besides the message type (as indicated by the Opcode), there are a number of defined fields that can be required or can be optionally present in a message, e.g. to encode arguments or return a result. This clause provides an overview of each of these fields, their descriptions, and some of the constraints on

their values. Each of these fields has a meaning that is consistent across message types, but the field may or may not be relevant or present in every message.

The definitions in this document of these fields and their types corresponds to a specific value of the CDAP Syntax Version that is established by CACEP. Since it is possible that CDAP can evolve, for example adding, removing, or modifying the meaning of fields or messages, we allow for the possibility by defining an integer corresponding to a version of the CDAP message abstract syntax. CACEP uses that integer in conjunction with the concrete syntax value to determine CDAP message processing and AE compatibility, and passes it to the AC in the ACSV, where it may be used to select behaviors such as an appropriate encode/decode implementation.

The value of the CDAP Abstract Syntax Version corresponding to this document is 0 (zero). Future revisions that would break compatibility with any previous version of the specification will be explicitly identified in any updated specification.

Any error in parsing or interpreting a CDAP message that is detected by the CDAP implementation itself (as opposed to being detected by an object method invocation) will result in an error per the error policy [POL-CDAP-ERROR] and will not result in invocation of an object method. A message that passes CDAP implementation validation will result in an object method invocation. Object methods may also detect errors; the error policy as well as object method definitions specify how those errors are handled.

Tables 2, 3 and 4 summarize the type and description of the different CDAP message fields in this version of the CDAP abstract syntax, and how they are used in specific messages.

Table 2 — Fields in messages — Summary

Field name in specification (standard abbreviation)	Type	Description
Filter	bytes	Filter predicate function to be used to determine whether an operation is to be applied to the designated object(s) or not. If the function returns TRUE for the object, the operation in the message is performed; if FALSE, it is not. The form and semantics of this field and how Result is determined are defined by the object policy, not defined by CDAP.
Flags	integer	Logical "or" of bit values representing Boolean variables that modify the meaning of a message when true.
InvokeID	integer	Unique identifier provided to identify a request, used to identify the subsequent associated reply.
ObjClass	string	Identifies the object class the ObjValue in the message.
ObjID	integer	Object Instance uniquely identifies a single object in an application's RIB. Either the ObjName or this value, or both, may be present, if the Object Model allows. If only a name is supplied in an operation, a corresponding ObjID may be returned in the reply, and that may be used in future operations in lieu of ObjName for the duration of this AC or for the object's lifetime if shorter.
ObjIDParent	String	Used to identify the object ID of the parent of an object, in conjunction with ObjNameParent, when a node is being created.
ObjName	string	Identifies a named object that the operation is to be applied to. Object names identify a unique object within an AC.
ObjNameParent	String	Used to identify the name of the parent of an object, in conjunction with ObjIDParent, when a node is being created.
ObjValue	structure	Value associated with the data field(s) of the object. May have a scalar, array, or compound object type and value.
Opcode	enum	Message type of this message, shall select one of the limited sets of message types described in this document, that selects the object method to call.

Table 2 *(continued)*

Field name in specification (standard abbreviation)	Type	Description
ResultReason	string	Additional explanation of Result.
Result	integer	The result of an operation, indicating its success, partial success in the case of some operations, or failure and possibly a reason for failure. All replies return a result, but, if allowed by the concrete syntax, the field need not be explicitly present when the result is unconditional success (zero).
Scope	integer	Specifies the depth that the operation is to extend beyond (or number of levels below, in a tree-structured object model) the designated object to which an operation is to apply (subject to filtering). If missing, or present and having the value 0, only the designated object is affected.

In Table 3, the presence or absence of a field in a particular message type is defined by one of the following treatments.

Table 3 — Legend for message field table

Symbol	Explanation
M	Mandatory field, shall always be present in a message of this type.
m	Field value that shall be provided explicitly or implicitly in a message of this type. Note that the concrete syntax in use [POL-CDAP-CSVersion] may define a missing field to satisfy this requirement by having a specific default value (e.g., zero), or the ObjClass may be unambiguously implied by the ObjValue field encoding (e.g., a string), or the ObjIDParent or ObjIDNameParent may be inferred from the ObjName per the object policy; in these cases, the mandatory value requirement is met and the field may be missing from the message encoding.
O	Allowed, but not required or validated by CDAP. Object models may further specify the use and meaning of the field.
C	Conditional, may be present as specified for each message type.
V	Allowed, presence is an object model option, but if present the field value is permitted to be validated by CDAP against the expected value (e.g. matching a field in a reply to the original request to validate that a reply is to the correct request) as a consistency check for error resilience and reservation for future semantic changes.
=	Field mandatory, value shall match that of InvokeID in associated request message (see message descriptions for usage of InvokeID).
(blank)	Not permitted, presence is considered an error and invalidates the message; the error will be processed per policy [POL-CDAP-Error].

Table 4 summarizes the situations under which each of the fields named in the leftmost column appear in the messages indicated in the rightmost columns. The "=" indicator in replies means that the value supplied in the message shall be present and match that from the corresponding request message, usually in the column immediately to the left, or to which the response or message applies, as for `A_CANCEL_READ`.

Table 4 — Message fields

Field name / Message	CREATE	CREATE_R	DELETE	DELETE_R	READ	READ_R	CANCELREAD	CANCELREAD_R	WRITE	WRITE_R	START	START_R	STOP	STOP_R
Filter	O	V	O	V	O	V	O	V	O	V	O	V	O	V
Flags	C	C	C	C	C	C	C	C	C	C	C	C	C	C
InvokeID	O	=	O	=	O	=	=	=	O	=	O	=	O	=
ObjClass	m	V	V	V	V	V	V	V	V	V	V	V	V	V
ObjID	R	V	R	V	R	V	V	V	R	V	R	V	R	V
ObjName	R	V	R	V	R	V	V	V	R	V	R	V	R	V
ObjIDParent	m	V												
ObjNameParent	m	V												
ObjValue	O	O			O	C			m	O	O	O	O	O
Opcode	M	M	M	M	M	M	M	M	M	M	M	M	M	M
ResultReason		O		O		O	O	O		O		O		O
Result		m		m		m	O	m		m		m		m
Scope		V	O	V	O	V	O	V	O	V	O	V	O	V

5.8.2 Opcode

This field selects the method to be performed by the recipient of the message. It shall identify one of the CDAP messages defined above, encoded as defined in the CDAP Profile's specified concrete syntax.

Messages are one of two types: request or reply. Each request message, named "X" has a corresponding reply message named "X_R" that is used to reply to it and return a result. With the exception of messages for which no reply is requested (see InvokeID), and READ/READ_R and CANCELREAD and CANCELREAD_R in an unusual situation (see CANCELREAD), request and reply messages occur in pairs. A request message eventually results in a corresponding reply message.

5.8.3 InvokeID

Message exchanges begin with sending a request that typically expects a subsequent reply message to be received, indicating the success or failure of the operation and sometimes including a result value. To be able to associate a reply message with the original request, each request message carries a tag value called the invokeID, and that same value is included in the corresponding reply message to identify it. If a reply message is desired, a non-zero InvokeID value shall be present in the request message. Otherwise, no reply will be generated in response to the request.

The InvokeID is also used by the CANCELREAD message that is used to cancel an outstanding READ request. In this case, the InvokeID provided in the CANCELREAD matches the value used in the associated READ request. See the CANCELREAD message description for further details of this usage.

The InvokeID has a non-zero integer value that shall be unique among all outstanding request operations. Other than CANCELREAD, a request with an InvokeID value that is already in use at the receiver will not result in a method call, and will result in an action performed per the error policy [POL-CDAP-Error]. Duplicate InvokeID values are detected before examining any other fields of a request message except Opcode, so other errors present in the message will in general not be detected and reported.

Applications may use fixed values for invokeIDs, may cycle through a small number of invokeID values, or may chose them from a larger space (the use of a larger space may provide better detection of protocol errors.) The InvokeID value zero is reserved to indicate that no InvokeID is being specified; in encoding

syntaxes that permit omitting fields from messages, a missing InvokeID field shall be interpreted as having a value of zero. There is no other restriction on the InvokeID value, and no other meaning is attached to the value by CDAP.

Upon generation of a request to which a reply is requested, the requesting CDAP implementation creates a transaction state machine for the invokeID to allow recognition of the returning reply, and sets its initial state to one that allows recognition of the appropriate response message. Whenever a reply message is received, the invokeID in the message is checked against all active invokeID state machines. If no match is found, the message is discarded, and a protocol error may be logged and reported. If a match is found, except for the case of an incomplete read response (see the description of the READ message), the state machine is destroyed, and the reply message is then further processed.

After InvokeID validation, receiving CDAP implementations may as a consistency check validate additional non-empty fields of a reply message against the request message that was sent with that InvokeID, per the "V" entry in the Message Fields table above. A mismatch represents an implementation error, such as mis-use of an InvokeID, or a message corruption that may be logged and/or otherwise be reported. Such a failure may generate an action per the error policy [POL-CDAP-Error].

When a zero or missing InvokeID value is being sent in a request message, the sending operation is complete as soon as the request message is sent, no state machine is created, and no Result or ObjValue will be returned to the requestor in a reply. This option is provided because if the application logic is not interested in the return value of a result, for whatever reason, it is more efficient to suppress the reply than to request one and then ignore it when it arrives. This option is usually not meaningful for a READ operation unless the only intended result of the resulting method call at the destination is a side-effect. It is also useful in the case when CDAP is used over an unreliable medium, as the message or its reply can be lost anyway.

Although CDAP is normally used over reliable connections, it is possible to construct applications using CDAP that do not require guaranteed message delivery. However, this significantly impacts the handling of InvokeIDs, as the loss of a reply message can result in a state machine waiting indefinitely for a response with that InvokeID value and permanently preventing reuse of that InvokeID value. Therefore, applications structured to operate over an unreliable transport should generally not request replies. Such usage of CDAP becomes a convention that shall be consistently followed by all objects and shall be specified explicitly for them in the object model.

5.8.4 ObjName, ObjID

Either an ObjName or ObjID, or both, reference a unique object that is the target of a message, as described earlier. Either or both may be present, as specified in the policies [POL-CDAP-OBJ-ObjRef] and [POL-CDAP-OBJ-OBJCREATE]. Specifying a null ObjName is the same as not supplying a name; there is no default. ObjName is an opaque string to CDAP. It is up to the object model to specify the syntax, semantics, and encoding of the name, CDAP only uses it in identifying the object. ObjID is an alternative way to identify an object; it can be more efficient than using a name, but they are otherwise equivalent for CDAP's purpose. A value of zero for ObjID is reserved to indicate that the value is unknown or not supplied; no other values are reserved, and the ObjID value is not otherwise interpreted by CDAP.

If a CDAP message addresses an object by ObjName without using an ObjID, a reply message may include an ObjID that can be used in place of the ObjName in subsequent operations to address the object. The application assigning an ObjID value is obligated to ensure that the ObjID value corresponds to the same object for the duration of the AC or for the lifetime of the object, whichever is shorter.

5.8.5 ObjNameParent, ObjIDParent

These fields can be used in a CREATE message to identify the object that is to be considered the "parent" of the object named by ObjName/ObjID. CDAP itself has no concept of "parent", other than transporting these fields, but the object model may. When used, these fields uniquely identify the targeted parent. In some object models, the name of the object being created (e.g. a fully-qualified name in a hierarchical naming structure) will provide sufficient information to infer the parent object, and these fields would not be needed. In other object models (e.g. one in which names are not used at all, only ObjIDs), these

fields can provide that structural information. The policy defining how these fields are used, if they are, is [POL-CDAP-OBJ-OBJCREATE].

5.8.6 ObjClass

The ObjClass field specifies the object class of the ObjValue field encoded in the message. The object that is the target of a request message, or the object generating a reply message, determines what class or classes of ObjValue can be interpreted or generated by that object. If an ObjValue is present in a message but no ObjClass is present, CDAP mechanisms will attempt to decode or encode the ObjValue according to the object class of the targeted object. If decoding fails, CDAP mechanisms will report an error, but if an ObjValue in a request message is decoded properly per the ObjClass but that object class is not accepted by that object, the object itself reports the error. In either case, an action will be performed per the error policy [POL-CDAP-Error].

The ObjClass string shall match the name of a class that is defined in the object model in the CDAP Profile [POL-CDAP-OBJ-CLASSES] policy. The class definition informs the conversion that is to be performed by CDAP between the value as encoded in the ObjValue field in CDAP messages and the value portion of an object of that class. The values used as ObjClass name strings are not defined or interpreted by CDAP, they are simply compared for equality with object class names that are defined in the object model. The CDAP implementation provides the capability to convert an object's data fields into and from the concrete syntax used by the CDAP Profile and ObjValue fields given the class of the object.

For new objects being created using a CREATE message, either the ObjClass field shall be explicitly present in the message and shall match a known object class name, or the [POL-CDAP-OBJ-OBJCREATE] policy shall specify how the type encoding implicitly or explicitly present in the message concrete syntax is used to infer a specific ObjClass value.

The [POL-CDAP-OBJ-Types] policy in the CDAP Profile defines the set of basic scalar types that can be used to construct classes. These shall be sufficiently rich to represent the full range of values of all the fields of the defined objects; CDAP implementations need not implement types that are not used by any of the objects that they can operate on.

5.8.7 ObjValue

The ObjValue field of a message encodes the value of the data portion of an object. For a `START`, `STOP`, `WRITE`, or `CREATE` message, it encodes a value to be made available to the object for writing. The ObjValue for a successful READ message is provided in the `READ_R` reply message, and an ObjValue may optionally be returned by some other reply messages. The ObjClass in the message in conjunction with the concrete syntax encoding defined in the CDAP Profile [POL-CDAP-CSYNTAX] defines how the value is encoded in the message. The CDAP implementation is responsible for being able to encode and decode every type/class data portion defined for this CDAP Profile in the [POL-CDAP-OBJ-Types] and [POL-CDAP-CLASSES] policies.

When a message is created, the CDAP implementation API may allow the application to request CDAP to create the ObjValue field with a subset of the object's data fields included explicitly, and if the CDAP Profile's concrete syntax permits it, CDAP will generate messages with only those fields, omitting all other fields (thus potentially shortening the message). When an object method is invoked by CDAP, it is provided with the decoded value from the message and a description of which fields were explicitly provided values in the message and which were not.

CDAP mechanisms will attempt to encode the value of an object into an appropriate form for the designated object class, per the concrete syntax, as an outgoing message is being constructed. If a value cannot be encoded into the target field type of the concrete syntax, an error will be returned to the application and no message will be sent. Similarly, if an incoming message contains a value that after parsing cannot fit without loss of information into the target data field of the object, the message will be treated as having a CDAP message encoding error and action performed per the error policy [POL-CDAP-Error]. No method will be called.

5.8.8 Result and ResultReason

The Result value is a signed integer, returned in response messages to indicate whether the requested operation was successfully performed. The value 0 (zero), which is assumed if the field is omitted from the message, is reserved and indicates unconditional success. The value –1 is reserved and indicates that the operation did not complete successfully, but does not guarantee whether or not some side-effect occurred. Other specific values that provide additional information may be defined in the CDAP Profile. All values less than zero indicate failure, and may encode a reason or reasons for the failure. All values greater than zero indicate non-failure, but may also convey other explanatory information about the results of the operation. The error policy shall define any values that are intended to have the same meaning by all implementations of the object model. Implementations shall only use values defined in the policy in determining an action to take after success or failure; all other success or failure values should be treated as if they were 0 or –1 respectively.

The ResultReason value optionally provides additional information about the result. Its contents are an application choice, and it is never mandatory. It may be returned in any reply, as desired by the applications. It has no CDAP-mandated format, but can, for example, by application convention, be a Unicode string that provides a human-readable explanation for a failure, or provide information for logging and tracking purposes. The form and usage of this field, if used, shall be described in the object method definitions in the object model (or in other CDAP Profile policies).

In low-trust ACs it is recommended that a minimum of information be provided to the other party. Therefore, a descriptive Result value or ResultReason should generally not be supplied if the reason for a failure can convey useful information to an untrusted party, such as invalid account name, invalid password, inadequate access privilege, existence of an object that the caller does not have the requested access to, etc. See also the description of the ACSV "Verbose" Boolean.

5.8.9 Scope and filter

CDAP is often used in an environment where the RIB data possesses a hierarchy or does grouping of objects (for example, "all flows, organized by destination"). It can sometimes provide a large saving of communication time and cost to operate on a set of objects using a single message, though such usage can also greatly increase the impact of any errors made. CDAP provides a basic way to encode a request to serially perform a specific method call on multiple objects with one message, though it is a choice of the CDAP Profile how and whether to permit these operations. Two fields are provided to implement this capability: Scope and Filter.

A message that includes the Scope and Filter fields names a single object per the object reference policy using its ObjName and/or ObjID; we refer to that object as the root of the operation. The remaining fields, particularly the Opcode, ObjClass, and ObjValue fields, specify the method to perform on each object; the ObjClass and ObjValue are provided to each method invocation. The Scope is a non-negative number indicating how many levels into the structure owned by the root, whatever that means in the object model, to propagate the operation. The value zero is reserved to indicate that only the target is to be affected, any other value indicates the additional depth into the structure. The simplest example is a (sub)tree, rooted at the target. The Scope value in this case is the depth of the nodes below the target to operate upon.

The Filter is a predicate that optionally restricts the operation to objects satisfying some condition. The Filter is a side-effect-free Boolean predicate, with syntax and semantics defined in the CDAP Profile. The Filter procedure is invoked for each object within the scope, providing the predicate with the object as an argument. If the predicate returns True, the method call is made to the object. If not, no call is made.

CDAP defines no other semantics for Scope/Filter. Any CDAP Profile that defines Scope/Filter operations shall define the meaning of Scope, the syntax and semantics of the Filter predicate, and define the semantics of each method call that is allowed and how return values, if any, are defined. This policy is [POL-CDAP-SCOPEFILTER].

5.8.10 Flags

The Flags field carries a set of Boolean flags that can modify the meaning of a message, encoded into an integer.

The F_SYNC flag may be set TRUE on request operations to indicate that if multiple values are being read or written or if multiple side-effects occur as an effect of the message, that they are requested to be performed in a synchronous, atomic manner, i.e. not interleaved with other operations that can be outstanding at the same time. Providing synchronous operations can be an expensive or impractical operation in some cases, so the default semantics of the order of execution of operations that affect multiple objects (possibly hidden, presented via a single virtual object) is that access or side-effects occur in an unspecified order. The F_SYNC option is useful for virtual objects whose methods shall access multiple underlying objects or for Scope/Filter operations. This is not always a meaningful request, and not all implementations can be capable of providing the requested coherency. The CDAP Profile shall define the behavior of methods on objects when this flag is TRUE in policy [POL-CDAP_ORDERING]. The Profile shall also define how the Result and Flags values in reply messages are set by objects if the F_SYNC request is known to have failed. The default is to ignore F_SYNC, unless otherwise specified.

The F_INCOMPLETE flag may be used in a READ request to indicate that the response to the READ is permitted to be incomplete and shall be used in a READ_R when the READ is incomplete and that the InvokeID should not yet be retired. F_INCOMPLETE is further discussed with the READ_R message, as described below.

Unassigned bits in the Flags field that are not defined in this document shall be ZERO in messages. A missing flags field in a message, if allowed by the concrete syntax, shall be interpreted as all-zeros. The unasserted value of any new flag defined in the future will be zero. This convention reserves the flags for use in future releases. A non-zero value of an undefined flag bit in a Flags field is a CDAP message encoding error and will be treated per the error policy.

5.9 Object identification in messages

An existing object shall be unambiguously identified in a message in order for the recipient of the message to identify its methods and data portion and to operate on it. Policy [POL-CDAP-OBJ-ObjRef] identifies how objects are to be unambiguously referenced using a combination or alternative of ObjName and ObjID. In the following list, references in messages to existing objects are described as being "legal object references" to distinguish them from "illegal object references".

A message with a legal object reference:

— unambiguously references a single object by its ObjName and/or ObjID per the object reference policy.

An illegal object reference includes one of the following:

— does not satisfy the policies (e.g. has neither ObjName or ObjID, or otherwise is missing a required identifying field);

— has both an ObjName and an ObjID, and they reference two different objects;

— the object does not exist.

The rules are slightly different when creating a new object, as described for the CREATE operation described below.

Messages containing an illegal object reference are rejected by CDAP prior to any method invocation. They may be logged or some other action performed per the error policy [POL-CDAP-Error]. Additional information about the cause of the failure may be encoded into the Result value and/or ResultReason fields.

5.10 CDAP message/method Ttpes

5.10.1 Object creation: CREATE(_R), DELETE(_R)

CREATE and DELETE messages are used to create new objects and to delete existing objects from the RIB of the receiving application. If the object referenced by the CREATE message does not exist and the message is otherwise valid, an attempt will be made to create an object with the designated ObjName and/or ObjID, per the [POL-CDAP-OBJ-OBJCREATE] object creation policy. Otherwise, the operation will fail and be treated per the error policy.

The object class of the created object is determined by the contents of the message, and/or the object's designated parent, as determined by the object creation policy. If an ObjValue is provided in a CREATE message and the value and its ObjClass (whether explicitly provided or implied by the type information inherent in its encoding in the concrete syntax of the message) are acceptable to the recipient CREATE object method and/or its parent, the new object will be assigned an initial value based on it. If no ObjValue is provided, the object may be assigned a class-dependent initial value if so-specified in the object creation policy or in the object class definition for the object's class.

If a CREATE addresses an existing object, the object creation policy determines whether the operation is considered an error and handled per the error policy, or whether the CREATE is allowed to simply write a new value to the object. A CREATE operation on an existing object cannot change its ObjClass, but may in the latter case change its value.

ObjIDs may in some object models be solely assigned by the application at the destination of a CREATE operation, and in those models shall not be specified in a CREATE message. If an ObjID is specified in the CREATE message but cannot be used for the new object for any reason, the operation will fail. If no ObjID was provided, one may be assigned in accordance with the [POL-CDAP-OBJ-OBJCREATE] policy and may be returned in the CREATE_R reply along with a success result.

Objects may be created and destroyed by an application at will in the application's own RIB, but if an object whose ObjID has been shared during an AC with the apposite application is destroyed. other than by a DELETE operation by the apposite, that ObjID value shall not be reused for a different object during that AC.

CDAP does not specify how a new object is to be fitted into any hierarchy that might be present in the destination application's RIB; this is a policy decision. The object model policy [POL-CDAP-OBJ-OBJCREATE] describes how the parent of newly-created objects is determined, e.g. by examining the ObjName to identify a containing node/parent, or by using the ObjNameParent and/or ObjIDParent fields.

Objects can be deleted by sending a DELETE message referencing the object. If the object is not found, or cannot be deleted for any other reason, a DELETE_R reply with a failure result will be returned (if requested). If the object has been deleted, a DELETE_R reply with a success return value will be returned (if a reply is requested).

5.10.2 Object Read: READ(_R), CANCELREAD(_R)

5.10.2.1 General

An application can fetch the current value of an object using a READ operation. If the specified object exists, and the InvokeID value in the message is non-zero, and the value can be obtained, the value is returned along with a success result in a READ_R message. If the object or value cannot be located or read but a reply is requested, then a READ_R message containing a failure result is returned.

It is not mandatory to request a reply. Sending a READ message with no or a zero-valued InvokeID will not generate a READ_R reply, but may produce a useful side-effect depending on the object.

CDAP does not define the semantics of including an ObjValue in a READ operation, but it is allowed. The default is to ignore it, but an object can treat it as an error or implement an operation using it, e.g.

summing the ObjValue into the object's current value before returning the updated value, or exchanging the previous value with the value in the ObjValue field. Any behavior other than ignoring the ObjValue shall be documented in the object model policy for an object or class.

The F_INCOMPLETE flag may be set in the Flags field in a READ request as an indicator to the receiving object that an incomplete READ_R is requested. This is a suggestion and may be ignored by the object unless other behavior is explicitly specified in the object model.

5.10.2.2 Incomplete READ_R

If the value of the object is too large to return in a READ_R reply message, or if the semantics of the object indicate that it will be updated in the future, a READ_R can indicate that it is a partial response and that the READ has not yet completed by setting the F_INCOMPLETE flag in the Flags field of the READ_R reply message. Thus, multiple READ_R replies with the same InvokeID can be received in response to a READ. This could be used to, for example, implement a Publish/Subscribe operation on an object, responding with changes such as object value or status (e.g. entries in a log file or a temperature reading). The InvokeID state machine created for the READ operation by the requester will remain active until a READ_R with matching InvokeID and the F_INCOMPLATE flag unset is received, or a CANCELREAD_R is received with the matching InvokeID as described below. Any number of incomplete READ_R replies can be sent prior to completion of a READ.

In the object policy [POL-CDAP-READINCOMPLETE], the object definitions or the general object model shall identify objects that may return an incomplete READ and describe the semantics of the operation.

5.10.2.3 Canceling a READ with CANCELREAD

The sender of a READ can cancel the outstanding READ before receiving a corresponding READ_R by sending a CANCELREAD with the same invokeID value originally sent in the READ request. A READ_R message or messages responding to the READ may already be in transit and will still be delivered, but no new READ_R messages will be generated after the CANCELREAD is received. When the CANCELREAD message is received, if the InvokeID corresponds to an active READ state machine, the outstanding READ operation at the destination is cancelled, the InvokeID is retired, and a CANCELREAD_R is returned using the retired InvokeID, indicating that the corresponding READ operation has been cancelled.

The fact that the READ may have already completed and the InvokeID state machine at the destination destroyed when the CANCELREAD arrives leads to a race condition that shall be handled at the requesting side: when the sender of the READ and subsequent CANCELREAD is awaiting the CANCELREAD_R response, it shall also recognize that a completed READ_R with the matching invokeID may be in-flight before the CANCELREAD is received and acted upon. If the sender of the CANCELREAD receives a completed READ_R, it will not receive a CANCELREAD_R message because the CANCELREAD would have arrived after the invokeID was retired and would be discarded because the InvokeID wasn't active. In this case, the READ sender can immediately destroy the InvokeID state machine upon receipt of the completed

5.10.3 Object Write: WRITE(_R)

An application can modify or set the value of an object by using WRITE. An ObjValue argument is normally included to provide the new value, though it may be meaningful for some object classes to have an "unset" value, which can be provided by omitting the ObjValue completely. An ObjClass field provided in the message (or provided implicitly by the encoding of the ObjValue in the concrete syntax) describes the class of the ObjValue, and together these are presented to the WRITE method of the object. If the object does not exist, or if it does not implement the WRITE operation on the object using the provided ObjClass, or if it cannot be written for any other reason, a WRITE_R reply message will be returned (if requested) with a failure result. If the object exists and the object considers the WRITE to be successful, a WRITE_R reply message will be returned (if requested) with a success result.

Objects may be active objects, as well as simply containers for passive data. Therefore, CDAP WRITE messages may have side effects, such as affecting a device or causing an operation to be performed.

WRITE_R messages may optionally return a non-empty ObjValue; the meaning of such a value is not defined by CDAP, it is determined by the object and shall be described in the object model for either the specific object or for its class. Depending on object definition, it might be the value of the object before the WRITE is performed (an exchange-value operation), the value after, or something else, such as a function of the previous value and the value provided by the WRITE (e.g. a running sum).

5.10.4 Object Stop/Start: START(_R), STOP(_R)

CDAP does not define semantics for the started/stopped status of an object, other than to recognize that "operational status" is a common property of many real-world objects. Conceptually, starting an object with START begins the process of making it operational, whatever that means. Stopping an object with STOP begins the process of taking it out of the operational state.

If it is meaningful to the object, an ObjValue can be provided in a START or STOP message. For example, this can designate the power level that a running system is to be taken down to on a STOP. Interpretation is up to the object and shall be described in the object model.

Although it is left to the object to decide when to send a START_R or STOP_R, generally the reply will be sent after the object has either failed to begin the operation or failed to reach the desired state, in which case an error result will be returned, or has reached the requested state, with the result indicating success or degree of success. The exact meaning of START, STOP, and any returned result is a property of the object and shall be described in the object model. If no result is needed, a reply need not be requested.

6 Policies

6.1 General

All policies shall be fully defined in a CDAP Profile, which provides a policy definition sufficient for implementation for every policy that differs from the default. The Default policies provided here should be used if there is no compelling reason to use a different policy.

6.2 POL-CDAP-CSYNTAX — Concrete syntax

6.2.1 General

This policy identifies how CDAP messages are to be encoded for transmission over a flow as a stream of octets ("bytes") in a given AC. The syntax to be used is specified by an integer value established by CACEP and made available to CDAP in the ACSV. The value that defines which syntax CDAP shall use, as provided in the ACSV, shall be selected from Table 5. Values are in hexadecimal, with Ascii equivalents enclosed in single quotes for reference when useful.

Table 5 — Values of CDAP concrete syntaxes

Syntax version integer value	Name	Description
8	GPB	Google Protocol Buffers™ [ref], see Annex A for definition
7B ('{')	JSON	JSON [ref], see Annex B for definition
61-7A ('a'-'z'), 41-5A ('A'-'Z')	HTTP	Reserved, shall not be used at this time pending definition
80-FF	Experimental	Available for experimental use before adoption as a convention in this specification
80-FE	ASN.1-x	ASN.1 encoding(s) where x will correspond to PER, BER, etc.
All other values	Reserved	Reserved for future, shall not be used

6.2.2 Default

The default policy is GPB, as defined in Annex A, selected by providing a Concrete Syntax Version value per the table above in the ACSV.

6.3 POL-CDAP-AUTH — Authentication

6.3.1 General

Authentication policy is used if CACEP establishes an identity or other credentials which are to be used by CDAP in validating permission to access objects.

6.3.2 Default

A Boolean variable, Authenticated, is identified in the ACSV.

If a non-null authentication policy is defined for this AC and has been successfully performed, the Authenticated Boolean value in the ACSV is TRUE (non-zero) and the credentials in the ACSV are valid, otherwise is FALSE (zero).

The default is no authentication, and an ACSV Authenticated Boolean value of FALSE. Authentication credentials in the ACSV are not valid, and this AC shall not be allowed to access or modify any RIB objects with non-public values or meaning, or access the RIB in any way that could disrupt normal operation of the applications.

6.4 POL-CDAP-ORDERING — Order of execution of method calls

6.4.1 General

The InvokeID mechanism enables replies to request messages to be returned in a different order than the requests arrived. This is an optional behaviour. Unless otherwise specified in this policy, all request messages will result in the execution of their corresponding object methods in the order in which they are received, and replies will be returned in the same order as their corresponding requests.

This policy also specifies the treatment of the F_SYNC flag in the Flags field of messages. It may be set in any message, but has no effect unless a specific behaviour is specified in this policy or in the description of an object that obeys it.

6.4.2 Default

Incomplete READ_R messages, CANCELREAD_R messages, and the final completed READ_R after one or more Incomplete READ_R messages have been sent have no specified order of return with respect to their corresponding READ requests or other replies. All other method invocations occur, and replies are returned, in the order that requests are received.

6.5 POL-CDAP-OBJECTMODEL — Overall object model definition

6.5.1 General

The overall object model is defined by [POL-CDAP-OBJECTMODEL]. The overall policy comprises several named sub-policies. Together, these policies describe the object naming convention, how objects are identified in messages, how objects are related, the defined atomic types and classes, how objects are created and initialized and for what purpose, and the initial set of objects defined for this model. A CDAP Profile may modify or replace these individual policies or accept the defaults.

6.5.2 POL-CDAP-OBJ-VERSION — Object model version

6.5.2.1 General

This value is used to select from among the available Object Models known to both members of the AC. The value has meaning only among compatible AEs within compatible applications, there is no CDAP-required or CDAP-defined meaning assigned to any specific value or encoding of this number.

6.5.2.2 Default

At AC initiation time, CACEP will negotiate between the two AEs and choose a proposed Object Model Version value ("Version", below) to be used for the AC. The Version value increases in numerical value any time a material change in behavior is made to any object or policy of the object model, so the maximum values generally provide critical information about the degree of compatibility of the AEs. If an AE chooses to communicate with another AE that proposes to use a different version level than its own maximum, it shall accept or modify its behavior as necessary to preserve compatibility and security.

AEs may accept multiple Versions, or a range of Versions, for example in order to provide a smooth upgrade path. AEs should maintain a list of Version values for prior known-insecure or known-problematic versions and reject the AC if such a version is proposed, unless the AE is able to thwart or compensate for the problem.

Version values are the result of combining a set of three integers that define a Version, represented in textual form as:

major.minor.fix

where

— "major" differentiates among incompatible versions,

— "minor" represents the version level of important changes, such as fixing serious defects or security-endangering problems, that should be considered when accepting the AC, but the AEs should otherwise be able to communicate,

— "fix" represents any bug or cosmetic change version that should not change compatibility, but may correct a less-serious defect or change AC-invisible behavior (e.g. logging of errors).

AEs shall agree a priori on the criteria for "serious" vs. "less-serious" changes for purposes of categorizing version changes and agree upon encoding of these fields into a Version integer and on the maximum values allowed for each field.

The default encoding is that the Version is treated as a 32-bit unsigned integer, with each of the fields encoded into 8 bit unsigned bytes, with the high order byte (“extension”) reserved for use as described below and "major", "minor", and "fix" residing in the next-higher to low-order bytes, respectively.

If the extension byte value is non-zero, the entire Version value is defined in an AE-specific way, and will be interpreted as agreed upon a priori by the AEs.

The initial value for the "major", "minor", and "fix" fields is zero. The most significant field in which a change of the corresponding type has been made is incremented upon release of a change of that magnitude of the object model. All lower-order fields are returned to zero when any higher-order field is incremented.

AEs shall reject ACs with any "major" value not implemented by the AE. AEs may choose to reject a proposed AC with a lower "minor" value than their own maximum, but if an AE accepts an AC with a lower "minor" value than its own maximum value it shall accept behaviors consistent with that earlier version.

6.5.3 POL-CDAP-OBJ-VISIBILITY — RIB objects visible to this AC

6.5.3.1 General

This policy describes how the set of ObjNames/ObjIDs within the overall Object Model to be made visible to this AC is selected. This may be the entire RIB, a virtualized RIB subset (e.g. selected by AC AE), or any other view over the RIB.

6.5.3.2 Default

All objects in the RIB are potentially visible, but attempts to access a specific object will generate an “object not found” error if the requestor does not have permission to invoke any method of the object. However, some other error may still be generated during the method invocation, e.g. if the object does not implement a specific requested method or if the requestor does not have sufficient privilege for the requested operation.

6.5.4 POL-CDAP-OBJ-NAMING — Object naming convention

6.5.4.1 General

The naming policy has several parts:

a) the syntax of the string used in the ObjName field of a message to identify an object, if names are being used;

b) the syntax of the names of objects, if different;

c) how to use the contents of the ObjName field to identify the object.

All of these shall be specified.

6.5.4.2 Default

All objects have a non-null name, as described in the syntax below. ObjName fields in messages are defined in a way that presents a tree-structured name space that defines the path through the tree to the desired object. The underlying RIB may or may not be tree structured, but the ObjName string presents that appearance by its syntax. The “/” (hex 2F) character is used to separate parent node names from children node names.

It is up to individual object definitions whether parent nodes are objects that provide methods or serve only as naming artifacts used to create the tree structure. The sequence of parent nodes leading to the final (rightmost) child name in the ObjName, in addition to that rightmost child name, uniquely identify

the addressed object in the RIB. The rightmost name is considered to be the node name of the object; a string that identifies that object relative to its parent(s) within the tree is a pathname to the object.

The syntax of the ObjName field of a message:

node_name ::= [UTF-8 characters excluding "/" and NUL]+

object_name ::= node_name

relative_pathname ::= [node_name "/"]* object_name

absolute_pathname ::= "/" relative_pathname

Unless specified otherwise in a policy, the ObjName string in a message may either be an absolute pathname, which describes the path from the root of the name tree that culminates in the named object as a leaf, or a relative pathname, which can begin at an intermediate parent node of the tree below the root, or may name the object itself.

A separate policy may define how relative pathnames are to be interpreted; for example, a "default parent" object could be defined to provide a string that holds a name-prefix that is to be prepended to any relative pathname to convert it to an absolute pathname. This can be used, for example, to shorten messages by shortening ObjName fields. Absent such a policy, relative pathnames are interpreted as absolute pathnames.

The NUL character (hex 00), if present in an ObjName field, terminates the field for purposes of object identification; it and any characters following the NUL will be ignored during the process of object identification, but the entire field is available to invoked methods. If an object examines characters in the ObjName field beyond the first NUL, its behavior shall be documented. The default is to tolerate and ignore such characters.

6.5.5 POL-CDAP-OBJ-ObjRef — Use of ObjName/ObjID to edentify objects

6.5.5.1 General

This policy describes how ObjName and/or ObjID are used in request messages to uniquely identify the object to which the message refers. In any policy that allows both to appear in the same message it is an error when they do not refer to the same object and will be treated as such using the error policy.

The policy for the similar ObjNameParent and ObjIDParent fields which appear only in CREATE messages is defined in the object creation policy [POL-CDAP-OBJ-OBJCREATE].

6.5.5.2 Default

The default behaviour is that either the ObjName or the ObjID, or both, shall be present in a request message to identify the addressed object.

6.5.6 POL-CDAP-OBJ-OBJCREATE — Object creation

6.5.6.1 General

This policy defines whether new RIB objects can be created via CREATE messages, how either the object name or object ID, or both, are determined using any or all of the ObjName, ObjID, ObjNameParent, and ObjIDParent fields of the message, and how the ObjValue in the message is used. Additionally, any general `CREATE` policies or requirements that apply to all objects shall be specified. Optionally, object class specific or object specific `CREATE` rules which vary from the default may be specified here.

6.5.6.2 Default

Objects can be created using the CREATE message, subject to restrictions that may be imposed by authentication and other properties of the AC described by other policies, and restrictions imposed by the semantics of the specific object to be created.

An object class shall be provided by the CREATE message, whether an ObjValue is present or not. The ObjectClass may be provided explicitly by the ObjectClass field or implicitly by the encoding of an ObjValue in the concrete syntax. Unless documented otherwise for that object in the object definitions or in the object model for that class, this will become the object class of the newly-created object.

An initial value may be provided in the CREATE message. The ObjValue in the CREATE message will be provided to the CREATE method of the object, which will validate it and either use it or return an error per the error policy. If no ObjValue is provided, the class may either reject the CREATE request if there is no sensible default, or provide one; if there is a default, it shall be documented in the object class description.

Objects that already exist may be the target of a CREATE message. In this case, the CREATE may use the value in the CREATE message to modify the value of the object like a WRITE, though the message will be processed by the CREATE method of the object so the semantics may differ (e.g. there may be restrictions on the ObjValue that are based on the current value). If this behaviour is different from WRITE semantics, the object class definition or object definition CREATE method shall document the differences.

A created object shall have a non-empty name, which shall be specified in the ObjName field of the CREATE message in one of two ways. The ObjName field may contain an absolute pathname, in which case neither ObjNameParent or ObjIDParent may be present, and the name of the parent is deduced from the pathname. Or the ObjName field may have the name of the object, with its immediate parent being identified using either the ObjNameParent or ObjIDParent field; either an ObjNameParent or an ObjIDParent, but not both, shall be present in this case. If the parent is explicitly specified, the absolute pathname of the newly created node will be the concatenation of the absolute pathname of the parent, a "/", and the contents of the ObjName field.

In general, object ID values are assigned by the owner of the RIB that the object is being created in so that it can manage its own numerical object ID space. Therefore, the CREATE message shall not specify an ObjID unless the object ID value provided with the object name has been agreed upon a priori, e.g. by being an initial object per policy [CDAP-POL-OBJ-INITIAL] or corresponds to an acceptable value or range as documented in the object model per policy [CDAP-POL-OBJ-CLASSES] and [CDAP-POL-OBJ-ObjID]. If an ObjID is present and is not assigned in accordance with such policy, it is an error and the message is considered incorrectly formed and will be handled per the error policy.

The location of an object in the RIB, i.e. either its absolute pathname or immediate parentage, or both, can have semantic implications. Therefore, when objects are created, a parent object proposed by the CREATE message may need to intervene in the creation process. Although this mechanism is hidden and cannot be explicitly invoked via CDAP messages, any object that needs to provide selectivity over the creation of its own children objects shall implement a CREATE_CHILD method. This method, when present in the parent identified by a CREATE message, will be invoked by CDAP logic to potentially create the child targeted by the CREATE message. The parent's CREATE_CHILD method is passed the CREATE message and can exercise its discretion on how and whether to actually create the object. If the parent does choose to perform the object creation, it will in general request that CDAP logic create the object in the RIB and will then perform any operations required to complete its initialization, which may include invoking the CREATE method of the newly-created object with the original CREATE message. The presence and definition of the CREATE_CHILD method shall be included in the definition of any object class that provides such a method.

An object need not have a CREATE_CHILD method to be the parent of other objects. In the absence of a CREATE_CHILD method in the parent, or if the parent name does not correspond to an actual object but is simply a place-holder in the name space, the CDAP logic will create the object in the designated location in the RIB and invoke its CREATE method with the contents of the CREATE message.

6.5.7 POL-CDAP-OBJ-Types — Scalar types

6.5.7.1 General

Class names are always string-valued, their names are assigned in the object model. Built-in scalar types may similarly have defined names that are strings that can be used as the ObjClass value when creating scalar objects. The class of the value in a message may in some cases be inferred from the ObjValue field of a message as described below, in which case an otherwise missing ObjClass value may be considered to be supplied.

Implementations are not required to support all the defined types if they are not used by any defined object or class, for example, even though values such as double-precision float and 64-bit integers may be defined, they may never occur in messages if all the implemented objects have no value fields that use them.

Some concrete syntaxes do not support a rich set of types (e.g. JSON and GPB do not explicitly encode integer precision), so implementations are responsible for ensuring that object values can be coerced into a message syntax unambiguously and parsed from a message unambiguously so that the value is communicated accurately between the sender and recipient.

6.5.7.2 Default

The fundamental scalar type names shown in Table 6 are reserved for use by objects and object classes as the ObjClass of a scalar value in a message and may be used in specifications of aggregate (array or structured) object classes. When creating scalar objects, the names of these built-in types may be used as the ObjClass, or in some cases as noted elsewhere, the ObjClass may be implicit in the encoding. When creating an object, if a value encoded in the `CREATE` message using the current syntax does not have a self-evident precision or signedness, and there is no pre-arranged policy for selecting such properties, implementations shall provide an explicit ObjClass in the message.

Table 6 — Pre-defined default scalar ObjClass names

Name	Description
U1	Unsigned 8-bit (1 byte) integer value
U2	Unsigned 16-bit (2 byte) integer value
U4	Unsigned 32-bit (4 byte) integer value
U8	Unsigned 64-bit (8 byte) integer value
U16	Unsigned 128-bit (16 byte) integer value
I1	Signed 8-bit (1 byte) integer value
I2	Signed 16-bit (2 byte) integer value
I4	Signed 32-bit (4 byte) integer value
I8	Signed 64-bit (8 byte) integer value
I16	Signed 128-bit (16 byte) integer value
F4	32-bit (4 byte) floating point value, IEEE-754 format
F8	64-bit (8 byte) floating point value, IEEE-754 format
F16	128-bit (16 byte) floating point value, IEEE-754 format
B	Boolean value (“TRUE”, “FALSE”, or numerically coded as FALSE = 0, TRUE = any nonzero value)
S	Variable-length string, UTF-8 encoded
D	Variable-length arbitrary binary data

6.5.8 POL-CDAP-OBJ-CLASSES — Defined classes

6.5.8.1 General

Every object class to be used in a message sent in the AC shall be fully defined, including exposed fields, defined methods and their arguments, side effects of method calls, method invocation order restrictions ("protocol"), and (if specified in [POL-CDAP-AUTH]) any authentication/access restrictions on each method and field. Class names are string-valued, assigned and documented in the object model.

6.5.8.2 Default

Objects may have any scalar type as their object class, using the reserved type names. No additional classes beyond the scalar types are defined by default.

6.5.9 POL-CDAP-OBJ-METHODS — Object methods

6.5.9.1 General

This policy defines how specific method implementations are assigned to objects, and any other method behaviours that are specific to the object model.

6.5.9.2 Default

By default, each Object Class implements a method call for each of the CDAP message types defined in 5.10 that is meaningful for objects of that class. Any message that attempts to invoke an unimplemented method is in error and will be reported per the error policy.

All objects with the same Object Class share the same methods. There is no default mechanism for selectively associating a different method with any specific object. Methods whose behaviour varies depending on specific-object properties, for example, the location of an object in the RIB, shall internally modify their behavior to accommodate such variation.

6.5.10 POL-CDAP-OBJ-ObjID — ObjID values

6.5.10.1 General

ObjID values may be pre-assigned, reserved, have a pre-defined order of use or generation, be assigned within a specified range of values, values or ranges of values that the other party is allowed to assign, or there may be other rules governing the ObjID values to be used in an AC. Any such rule shall be documented in this policy. See also the [POL-CDAP-OBJ-OBJCREATE] policy for restrictions on ObjID values in CREATE messages. As documented elsewhere, an ObjID value may not be reused within an AC for a different object unless both parties are aware that the earlier object with that value has been deleted.

6.5.10.2 Default

ObjID values shall fit within a non-zero 32-bit unsigned integer. ObjID values are either assigned a priori and known by both ASs to refer to the same object, or otherwise shall be assigned by the AS that creates the object in its own RIB in response to receiving a CREATE message. There are no reserved or pre-assigned object ID values by default, other than those that may be assigned to initial objects per policy [POL-CDAP-OBJ-INITIAL]. All AEs shall use such initially assigned ObjID values for those objects.

Object Class definitions defined by policy [POL-CDAP-OBJ-CLASSES] may also pre-define ranges or conditions on ObjID values for use by objects of that class, which may include allowing ObjID values meeting those restrictions to be provided in CREATE messages. Those policies shall be defined consistently so that object id values are assigned in a non-conflicting and unambiguous way.

6.5.11 POL-CDAP-OBJ-INITIAL — Pre-defined objects

6.5.11.1 General

This policy provides the set of objects that can be assumed to be present at the beginning of an AC prior to any operations. This includes the defined name and/or object id, object class, parentage if not defined by the name or object id, any methods that differ from the class default, and initial value if different from the default for the object's class.

6.5.11.2 Default

None at this time.

6.6 POL-CDAP-ERROR — Error handling and return values

6.6.1 General

This policy describes how error conditions described in this specification, as well as errors arising in the execution of object methods, are handled. It catalogs the Return values used to describe successful and unsuccessful object operations. It defines the conventions for use and may provide a partial or complete dictionary of ResultReason strings corresponding to specific errors.

Error Return values returned in the Result field of a reply shall use values compatible with Table 7 to indicate success or failure. Any specific values returned from an implementation may be defined in the Policy. Specific failures can communicate additional information about the cause of the error that may be used in diagnosing the cause of the problem. Receipt of specific values as defined in the Policy may be used to select a corresponding action upon receipt. Implementations receiving an unrecognized specific value should treat it as equivalent to –1 if negative, or 0 if positive. If an AC is established with a low-trust application (not defined in this specification), or for any other reason, applications may choose to return only –1 and 0 values in lieu of specific ones.

Table 7 – CDAP return values

Value	Meaning
>0	Specific success
0	General Success
–1	General Failure
<–1	Specific Failure
–9999 to 9999	Reserved for CDAP Profiles, values outside this range are available for application use

6.6.2 Default

The policy may use different methods to report different types of errors and may choose to categorize handling into categories such as those in Table 8 (which is suggested, not mandatory).

Table 8 — CDAP error types

Class	Error type	Description	Examples
1	Communication Failure	Errors upon attempting to send or receive a message	Flow timeout, error on send, error on receive
2	Message Parse Failure	Message cannot be parsed per the concrete or abstract syntax	Missing or invalid OpCode, incorrect-length message, syntax or other error in encoding, corrupted SDU
3	CDAP Usage Error	Message contains one or more fields that violate a concrete syntax, CDAP limit, or other requirement of CDAP	Attempt to consume too many InvokeIDs, illegal field(s) in message for its type
4	Value, Type, or Range Error	A field contains a value that cannot be coerced to fit the type of the targeted field of an object, or is otherwise out of the specified range	ObjClass in message incompatible with addressed object, InvokeID out of specified range, integer value too large for target field, structured object value in wrong encoding or missing mandatory field(s)
5	Semantic Error	A method cannot be found or a method has reported that there is a semantic problem with the message	Inconsistent field values, value out of required range, inappropriate operation for object, unspecified error occurred during execution of method
6	Warning	A potential problem has been detected, but operation is not (yet) impacted	A limit (e.g. InvokeID limit) is being approached, an excess of messages is being received, a potential implementation error has been detected
7	Informational	Statistics, spontaneous events, information that could be useful in logfiles	Authentication success and time, flow statistics (e.g. response times), resource usage
8	Debug	Information not normally used in operation, but useful for finding problems	Logging of output of message parser, program flow markers, software revision and status information

Errors of class 1 can cause CDAP to request CACEP to abandon the AC, logging the event locally if possible. Implementations may choose to delay attempting to re-establish communication to the other application if this event occurs an implementation-defined number of times within an implementation-defined period.

All errors that cause a message to be ill-formed or uninterpretable (class 2-3) should cause CDAP to log the error locally, log it via a generated CDAP message to the other application if there is a means to do so, and may request to CACEP that the flow be closed. If an AE chooses to continue after such an error, unpredictable additional errors may result, so implementations should terminate an AC after a policy-set number of such errors occur in the AC.

All errors of classes 4-6 that occur in processing messages that request a reply should be replied to with an appropriate value in the Result field of the reply message, possibly then sending an additional error logging message to the other side if there is a means to do so, possibly followed by terminating the AC. All errors of class 4-6 in messages that do not request a reply may be logged via a generated CDAP message to the other side if there is a means to do so, possibly followed by terminating the AC. Errors should be logged for offline analysis.

All spontaneously generated messages of any level, including Informational and Debug messages, classes 7 and 8, may be logged via a generated CDAP message to the other side if there is a means to do so. If the amount of traffic so generated can result in exhaustion of InvokeID values, the reporting messages shall not request replies, or shall limit the number of InvokeIDs consumed to keep the number below the limit. If the amount of traffic sent would be so high (greater than an implementation-dependent rate) that it can result in a denial of service to normal ongoing message exchanges, the message generation shall be stopped or limited by discarding excess traffic. If any such traffic is limited by discarding, a periodic message of the same class indicating how many messages were discarded should eventually be sent.

Table 9 defines the followingSpecific Result values.

Table 9 — CDAP specific result values

Result Name	Value	Description
	< 0	All values less than zero indicate failure. Side effects may have occurred (application specific).
	< –9999	Reserved for application-specific errors.
	> 0	Non-failure. The value provides additional information.
	> 9999	Reserved for application-specific degree-of-success indicators.
R_SUCCESS	0	Unconditional success. The operation was performed as requested, and no errors were encountered.
R_SYNC_UNIMP	1	A sync operation could not be performed synchronously as requested, but the operation was performed successfully on a best-effort basis.
R_FILTER_FALSE	2	An otherwise-correct operation specified a non-null filter, and the filter returned FALSE for all addressed objects. No operation was performed on any object.
R_FILTER_MIXED	3	An otherwise-correct operation specified a non-null filter, and the filter returned FALSE for one or more objects, and TRUE for one or more objects. No operation was performed, and no value is returned, for the object(s) for which the filter was false. The operation is performed as requested for the object(s) for which the filter was true.
R_FAIL	-1	An operation failed for an unspecified reason. (This generic value should be returned in situations where the application does not completely trust the requesting application, in order to avoid providing it with useful information. No resultReason should be provided in that situation.).
R_OS_ERR	-2	An operation resulted in an error from the operating system, for example because of a file system error. If available, a description of the error is in the ResultReason. (Note that OS values are not standardized, so reporting them may have limited value).
R_OBJNOTFOUND	-3	The supplied ObjClass/ObjName pair, or supplied ObjInst, does not correspond to a known object.
R_OBJBADID	-4	The supplied ObjName does not correspond to the supplied ObjID value.
R_CLASSNOTFOUND	-5	The supplied ObjClass does not correspond to a known class.

6.7 POL-CDAP-InvokeID — Convention for assigning InvokeID values

6.7.1 General

This policy guides the assignment of values to InvokeIDs, and any semantics associated with the value of an InvokeID. The value zero represents no InvokeID and is therefore never an InvokeID value; the allowable range of valid InvokeID values and any restrictions or method that shall be used for their assignment, or any restrictions on the number that may be active at once, shall be specified. Receivers of an InvokeID value other than zero shall not recognize that value as having a specific meaning. If there are specific values of InvokeID that are significant for any reason to the requester (e.g. always used for a dedicated purpose), they may be specified in this policy for documentation purposes, but the recipient shall not take advantage of that knowledge.

6.7.2 Default

InvokeID values are assigned a value other than zero that can be encoded in a 32-bit unsigned integer. Implementations may limit the number of InvokeID state machines they implement for incoming operations, and thus the number of unreplied messages, to the number sufficient for their own operation. This may be a value as small as 0, implying that a reply is immediately returned for all

messages requesting a reply before the next incoming message is processed. Values have no implied semantics, and there is no required process for assignment of InvokeID values to requests.

6.8 POL-CDAP-READINCOMPLETE — Use of incomplete READ_R

6.8.1 General

This policy defines general rules for when an object or class of objects may return a READ_R with the F_INCOMPLETE flag set, and how to interpret the value and result in such messages. Generic rules that apply to all objects whose behavior is not specifically documented shall be stated in this policy.

The [POL-CDAP-InvokeID] policy may specify a lower limit on how many outstanding unreplied requests are allowed to be active, and may place a separate limit, which shall be no larger, on the number of unreplied messages that may be outstanding at any instant by objects repeatedly using incomplete READ_R replies.

For objects that may return an incomplete READ_R, each such object shall specify its behaviour if it differs from the default.

Requestors who receive an incomplete READ_R can at their discretion return a CANCELREAD response at any time. Responding objects or classes may not impose any other requirement on requestors.

6.8.2 Default

Objects or object classes that may return incomplete READ_R messages shall be documented to do so, and the conditions under which they do so shall be documented. Such objects should document the frequency (maximum and minimum), conditions that will cause a READ_R reply, and what conditions, e.g. an error or reaching a maximum number of incomplete READ_R replies, will cause the READ to complete.

Requestors may set the F_INCOMPLETE flag in the Flags field of a READ request message to explicitly request that such an object send incomplete replies. Responding objects may ignore the request and return a single READ_R reply.

Objects representing collections of objects, such as array-valued objects, may return any number of their elements in a succession of zero or more incomplete READ_R messages, followed by a completed (F_INCOMPLETE flag unset) READ_R message containing the final element(s), if any. Unless otherwise specified, array elements are returned in the messages in their ascending order in the array; all elements are sent, from first to last. The object may not be guaranteed to be unchanging during the sending of the messages, since sending can take arbitrarily long, so this operation should not be used unless that behavior is acceptable or the object definition provides a different defined behavior.

Any undesired incomplete READ_R reply returned from an object may cause the recipient to undertake an error handling action. A recommended action is to respond with a CANCELREAD request with a negative Error value, which allows the requestor to cancel its InvokeID state machine for the original READ request immediately and provides it with a reason for the cancellation.

6.9 POL-CDAP-SCOPEFILTER — Scope and filter policy

6.9.1 General

This policy defines how Scope and Filter are to be interpreted and applied. The policy shall define how the Scope variable is to be interpreted in the context of the object model. For a hierarchical RIB model, the scope shall be defined as the number of levels of the tree below the addressed node that are in scope, but for any other model a complete definition shall be provided in the policy.

The Filter policy shall define the syntax and semantics of the language used in the filter function, and how the procedure is encoded in the field.

6.9.2 Default

No defaults are defined.

6.10 POL-CDAP-ACSVContents — ACSV contents

6.10.1 General

This policy defines the contents of the ACSV. Every field, its type, the meaning of the field, and any defined values for the field, shall be provided. If there are requirements on specific values in the ACSV, they shall be specified. The ACSV itself is not communicated between applications, it is used solely by the application logic, but in order to provide the mandatory contents, specific information shall be included in the CACEP exchange and thus the [POL-CDAP-ACSVcontents] policy is a source of requirements for a compatible supporting CACEP implementation.

6.10.2 Default

The flow underlying the AC that is referenced in the ACSV shall provide in-order, gap-free, low bit-error-rate transfer of SDUs.

dAE name (string)	The name of the opposite-end AE.
dApp name (string)	The name of the opposite-end application.
sAE name (string)	The name of this AE.
sApp name (string)	The name of this App.
MaxSDULength (int)	The maximum length SDU that can be sent or received on the flow.
Authenticated (Bool)	Whether the authentication policy succeeded.

7 CDAP context notes

7.1 General

Clause 7 collects suggestions for use of CDAP that are not mandated but can lead to greater commonality of implementations.

7.2 RIB Daemon model

Applications can implement CDAP in many different ways. While not mandated or explicitly implemented in any way by the CDAP protocol, the RINA Reference Model assumes that CDAP operations are generated and interpreted by a “RIB Daemon” component of the application that also manages operations on RIB data objects. In this model, rather than being explicitly requested by applications, CDAP operations are typically generated as a side-effect of operations that an application performs on its RIB, onto which the RIB “view” is provided. The “view” is typically a combination of a subset of the application’s RIB objects and synthesized pseudo-objects.

In this model, the RIB Daemon is responsible for maintaining the values of RIB variables with the desired degree of consistency for each object, ideally transparently in the background. Thus, depending on the freshness requirements for data, and/or whether side-effects are to be created by an access, an internal data access by an application may result strictly in operations on local data, or may generate the need for the RIB Daemon to perform CDAP protocol operations immediately or eventually. For example, a hypothetical “RIBread(object)” operation on an object whose value is not cached or whose locally-held value is stale might cause the RIB Daemon to generate a READ CDAP operation. The API call

can delay return until the READ_R reply is received, or can return immediately with the requested value being made available later via an event.

Many other implementation approaches are viable; there is no mandated CDAP implementation API.

7.3 Distributed applications

The CDAP protocol can be used to create a distributed application, in which the objects stored in the RIB of an application instance represent its view of, and/or its portion of, the complete distributed RIB of the distributed application. The local view of the distributed objects' coherency, consistency, and timeliness are under the explicit control of the application by manipulating those properties of the object in the RIB, rather than by programming using explicit communication operations in the application. In this programming model, the CDAP and AE are not necessarily explicitly visible to the application, and the application will not use those communication API's directly. Its view of the distributed RIB is solely through its own local data objects.

ASN.1 syntax of CDAP

```
CDAP DEFINITIONS AUTOMATIC TAGS ::= BEGIN
CDAP-Message ::= SEQUENCE {
   opCode CDAP-OPERATION.&opCode ({AllOperations}),
   invokeID INTEGER DEFAULT 0,
   opData CDAP-OPERATION.&MessageType ({AllOperations} {@opCode})
}

OpCode ::= ENUMERATED { -- list of possible operation codes
          create,
          createResponse,
          delete,
          deleteResponse,
          read,
          readResponse,
          cancelRead,
          cancelReadResponse,
          write,
          writeResponse,
          start,
          startResponse,
          stop,
          stopResponse
        }

CDAP-OPERATION ::= CLASS {
   &opCode OpCode UNIQUE,
   &MessageType
}
WITH SYNTAX {
   OPCODE &opCode MESSAGE &MessageType
}

AllOperations CDAP-OPERATION ::= {
   InvokeOperations |
   ResponseOperations
}

InvokeOperations CDAP-OPERATION ::= {
   createOperation |
   deleteOperation |
   readOperation |
   cancelReadOperation |
   writeOperation |
   startOperation |
   stopOperation
}

ResponseOperations CDAP-OPERATION ::= {
   createResponseOperation |
   deleteResponseOperation |
```