AvesTerra™

A Framework for Global-Scale Knowledge Orchestration

Integration and Application Library (Avial®)

Version 4.0

December 2022

GEORGETOWN UNIVERSITY

V4.0 – December 2022

Table of Contents

Preface

This document is the fourth in a series of companion documents that collectively describe the AvesTerra system for global-scale knowledge orchestration and analytic interoperability. The list of key AvesTerra documents include:

- AvesTerra: FOUR-Color Framework

  The *FOUR-Color Framework* document provides an overview of the original principles and architectural aspects of AvesTerra's approach to knowledge sharing and analytic interoperability, including a description of the end-to-end system structure and each of the main four layers of the design. AvesTerra is the first major effort to make use of the full spectrum of FOUR-Color architectural principles.

- AvesTerra: Hypergraph Transaction Protocol (HGTP)

  The *HGTP* document provides a specification of the communication protocol used for all AvesTerra interactions including between AvesTerra clients and AvesTerra servers, and between AvesTerra servers with their peers.

- AvesTerra: Application Programming Interface (API)

  The *API* document describes the application programming interface constructs of the AvesTerra system. This interface is intended for systems level developers needing to interact with AvesTerra servers at the core level.

- **AvesTerra: Integration and Application Library (Avial)**

  The *Avial* document (presented here) provides the definition and technical details of the layer of standardized integration and application services built directly atop the AvesTerra API. This document is intended for developers integrating new data sources, adapters, analytics, tools, and application components into the AvesTerra ecosystem.

Acknowledgement

This paper is a result of lengthy technical conversations with an extraordinary community of researchers, analysts, and computational science talent spanning the Nation over the past two decades. Special thanks to the original contributors including Lawrence Livermore National Laboratory, Oak Ridge National Laboratory, Pacific Northwest National Laboratory, and the Raytheon Company for their intellectual contributions to this effort, to the Defense Advanced Research Projects Agency (DARPA) for their sponsorship of this work under the SIMPLEX program via contract #N66001-15-C-4044, and to LEDR Inc. for their enormous design, implementation, and testing feedback. The views contained within are solely those of the author and do not necessarily reflect the opinions or positions of Georgetown University or any collaborator or sponsor organization involved.

For additional information, contact:

J. C. Smart, Ph.D.
AvesTerra Chief Scientist
Research Professor
Department of Computer Science
Office of the Senior Vice President for Research
Georgetown University
37th and O Street, NW
Washington, D.C. 20057-1241
smart@georgetown.edu
(443) 745-5876

Overview

The AvesTerra Integration and Adaptation Library (Avial) provides a standardized set of functions for building applications and adapter tools and services within the AvesTerra knowledge orchestration framework. This framework is used to design and implement highly-scalable, decentralized knowledge-based applications and services. Avial is a layer of software constructed directly upon the AvesTerra Application Program Interface (API) as shown in Figure 1. The AvesTerra API provides unified access to the distributed AvesTerra knowledge space. Avial uses the resources of its layer to provide a set of higher-level services to application layer components above (e.g. ATra, Avert, Avian, and Orchestra). This organization helps simplify and accelerate software development by reducing the implementation burden that might otherwise be required to construct applications directly upon the AvesTerra API.

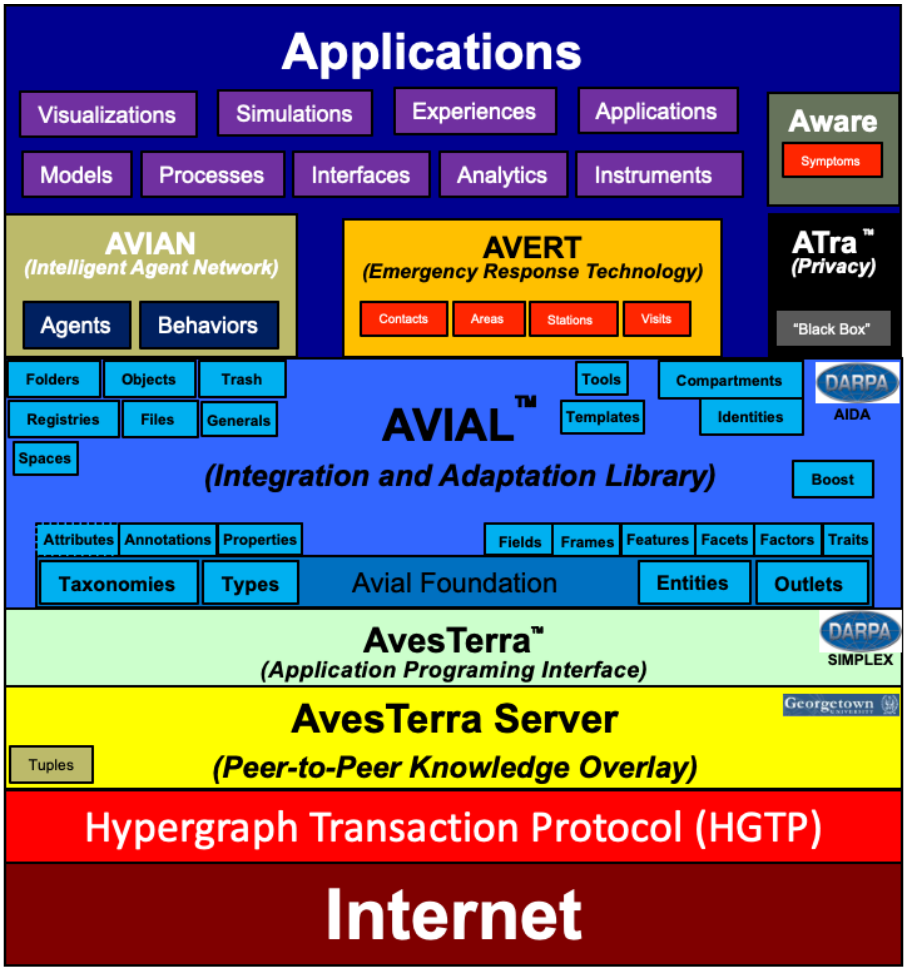

Figure 1. The AvesTerra Layered Design

The Avial Model

The centerpiece of Avial's services is a common model for organizing and representing information in the AvesTerra knowledge space. A graphical overview of this model is shown in Figure 2.

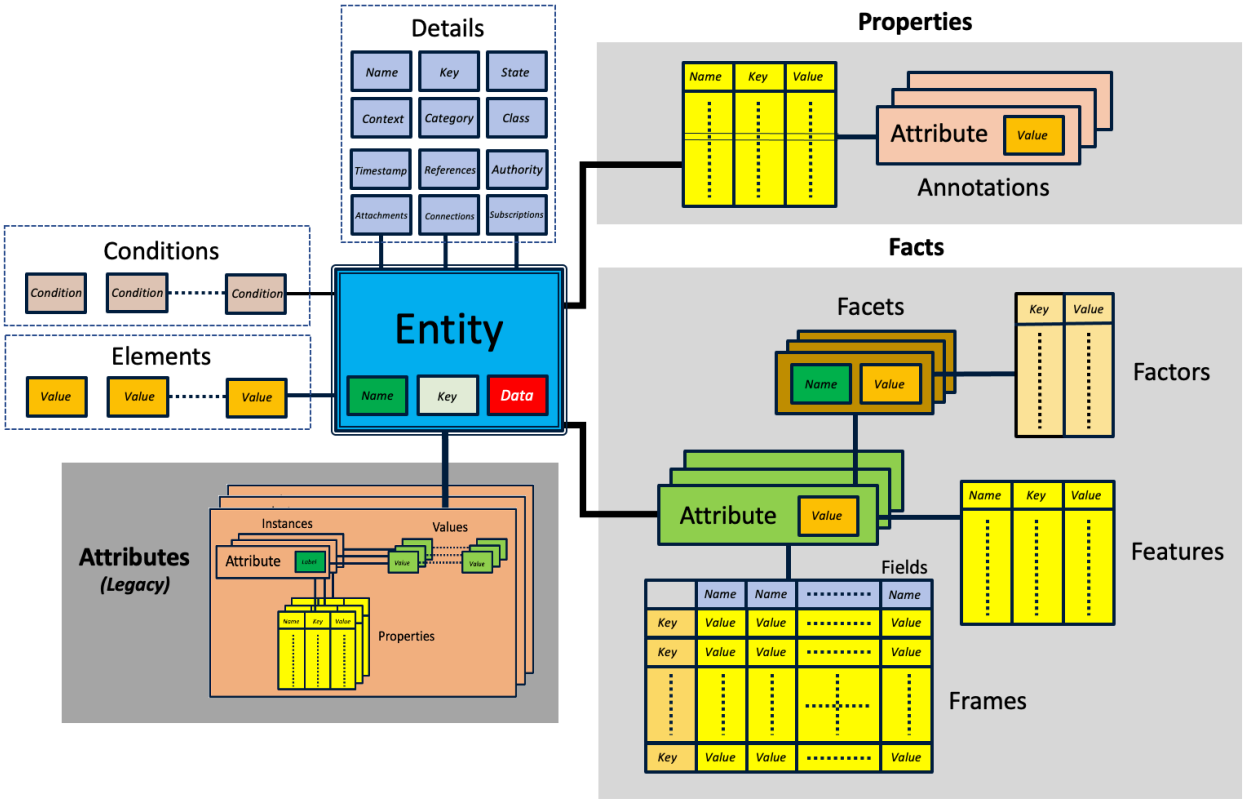

Figure 2. The **Avial** Model

The fundamental organizing principle within Avial, as supported by the framework layers beneath, is the notion of an ***entity***. An entity is a representation of an element of knowledge which typically possesses some distinct and independent existence in the physical world (e.g. a person, a place, a thing, etc.) or abstract world (e.g. a policy, concept, relationship, etc.). Avial entity types are governed by category, context, and class taxonomies, all of which are also represented within the knowledge space. These taxonomies are enumerations of all the possible entity variants that can represented within the knowledge space. These enumerations are extensible and, in general, intended to be continuously backward compatible as the taxonomy expands and evolves.

Associated with every entity is a *name*, used is for every-day labeling and identification, and a unique *key*, to aid entity storage and retrieval. Also associated with every entity is the ability to optionally store a volume of arbitrary data (e.g. a binary data file). Data may be read and written via a set of bulk transfer operations. (Note: the Avial system itself maintains no semantic understanding nor syntactic interpretation of the data being transferred.)

Avial includes a set of library functions for accessing all of these items. In support of general knowledge representation, Avial offers two powerful organizing constructs: the ***properties*** and ***facts***. These two constructs, each referred to as a model aspect, replace Avial's prior legacy attribute structure (i.e. Version 3 and earlier).

Properties

Properties are collections (i.e. a list) of name/value pairs, each with an optional key field, that can be associated with an entity. Within a single entity, names can be any arbitrary text string of acceptable length (i.e. < 256 UTF-8 characters). Keys similarly are character strings, however, they must be unique across all properties within an entity's property list. The value portion of a property can be any of Avial's value types. At present, Avial supports 22 base types:

- Null
- AvesTerra (reserved for server use)
- Boolean (true or false)
- Character (single 8-bit character)
- String (array of 8-bit ASCII characters)
- Text (array of UTF-8 characters)
- Integer (64-bit integer)
- Float (64-bit float)
- Entity (128-bit AvesTerra unique ID)
- Time (64-bit encoding of Unix-based time)
- Web (Internet URL)
- Interchange (JSON-based exchange)
- Data (8-bit binary data sequence)
- Exception (AvesTerra error encoding)
- Operator (AvesTerra Tuple-Machine instruction encoding)
- Function (AvesTerra Tuple-Machine function encoding)
- Measurement (value, units, prefix, precision, and accuracy encoding)
- Locutor (Avial "locution locator")
- Authorization (128-bit Internet UUID)
- Duration (delta time, in seconds)
- Aggregate (key-based collection of Avial value types)
- Array (array of Avial value types)

Each property in a property list may have zero or more ***annotations***. An annotation is an attribute/value pair. Attributes are defined by the Avial attribute taxonomy; an extensible enumerated list, similar to the category, context, and class taxonomies. All annotations for a specific property must have a unique attribute, unless the attribute is null. Values can again be of any of Avial's base value types. Properties with annotations are a very convenient and efficient knowledge construct. With a single network call, an entire annotated property list can be efficiently retrieved. Properties are particularly useful for creating registries where fast look-up is required.

Facts

Whereas the property construct is tuned for knowledge orchestration *efficiency*, the fact construct is designed for knowledge orchestration *expressivity*. Facts provide a means for capturing and

representing a wide variety of complex semantic structures including tuples, graphs, hypergraphs, and ultragraphs. A fact is an attribute/value pair where each attribute, if not null, must be unique. Valid attributes are again dictated by the attribute taxonomy. In turn, each fact can have zero or more facets, features, fields, and/or frames. Facets are name/value pairs where each name, if not null, must be unique. Every facet of a fact can also have zero or more factors. A factor is a key/value pair where each key, if not null, must be unique. Facets provide a convenient way to represent multiple named instances of a fact attribute. With facets uniquely identified by their name, factors provide a means to add uiniquely keyed values to facets. When the multiple instance notion is not needed, features are used allowing full name/value pairs with an optional key to be associated with a fact. Each key of a feature again, if not null, must be unique. Features behave much like properties, but are associated with a fact attribute versus an entity itself.

Finally, the combination of fields and frames provide a general-purpose means for building tabular structures of arbitrary length (rows) and width (columns). Fields essentially serve as a "column" guide for frames. Specifically, fields are a collection of name/value pairs where each name, if not null, must be unique. Adding (removing) a name to (from) a field list results in a corresponding value being inserted (removed) from every frame. For the insertion case, the named value of the specified field serves as the default value that is initially inserted into each frame at the corresponding column position. Frames thus mimic their list of fields, but allow any valid Avial value to be store at any column position. Together, fields and frames provide access to any cell within the table structure using either their row/column position, or a unique name and unique key.

Implementation

Each of the Avial list structures above can be accessed in two different ways: either index-based or map-based. All of the Avial standard adapters accomplish this using a single generic data referred to as an "indexed map". This data structure allows items at any position in the list to be accessed using an *index*. Indexes are positive integers that range from 1..*n* where *n* is the length of the list. Avial lists can be quite long, with the maximum length typically limited only by the memory and storage constraints of the underlying infrastructure. In addition to an index, list items can also be accessed using either a unique attribute, unique name or unique key, depending upon the specific data structured. The attribute, name, or key is used to hash to the corresponding list value. This index-mapped data structure allows many different types of semantic structures to be built while still maintaining a high degree of implementation efficiency.

Each of the data structures in the Avial model hierarchy are identified using an *aspect* parameter. Aspects are enumerations defined by the aspect taxonomy, similar to the other Avial enumerations. For each of the Avial's eight aspects (properties, annotations, facts, facets, factors, features, fields, and frames), 19 distinct operations (*methods*) are defined. Methods are once again enumerations listed in the method taxonomy. Index-based operations are performed using the **Insert**, **Remove**, **Replace**, and **Find** methods. Mapped-based operations are performed using the **Include** and **Exclude** methods. The **Set**, **Get**, and **Clear** methods typically have map-based behaviors, but may also affect other aspects in the aspect hierarchy. For examples, setting a facet value will results in a fact being created if the specified attribute did not already exist on the entity. The **Count**, **Member**, **Name**, **Key**, **Value**, **Index**, and **Attribute** methods all refer to respective utility functions for accessing different model aspect information. Lastly, the **Purge**, **Sort**, and **Retrieve** methods are plurality operations which can involve multiple model aspects at one time.

Application access to any of the Avial model aspects is provided through a set of function libraries included in the Avial software. The implementation of each function in the library is typically quite similar and very simple. In general, each library function need only make a single call to the Avial *invoke* method that is built directly atop and defined in the AvesTerra API. The difference between each library function call, however, is the particular parameters that are passed. These parameters include the specific method and aspect, the name, key, and/or value parameters, and any additional needed index parameters. As the Avial model aspect hierarchy can be up to three levels deep (e.g. Fact.Facet.Factor), the index, instance, and offset parameters are used to convey the appropriate list positions as they may be needed . Table 1 contains a complete list of the arguments for all 161 of the Avial model library functions. Fortunately, the implementation of each of these functions is very thin (e.g. one line) and thus easy reproduce for many different programming language environments.

Method	Properties	Annotations	Facts	Facets	Factors	Features	Frames	Fields
Insert	N,K,V,X	A,K,V,X,I	A,V,X	A,N,V,X,I	A,N,K,V,X,I,O	A,N,K,V,X,I	A,K,X,I	A,N,V,X,I
Remove	X	K,X,I	X	A,X,I	A,N,X,I,O	A,X,I	A,X,I	A,X,I
Replace	N,K,V,X	A,K,V,X,I	A,V,X	A,N,V,X,I	A,N,K,V,X,I,O	A,N,K,V,X,I	A,K,X,I	A,N,V,X,I
Find	N,V,X	K,V,X,I	V,X	A,V,X,I	A,N,V,X,I,O	A,N,V,X,I	A,N,V,X,I,O	A,V,X,I
Include	N,K,V	A,K,V	A,V	A,N,V	A,N,K,V	A,N,K,V	A,K	A,N,V
Exclude	K	A,K	A	A,N	A,N,K	A,K	A,K	A,N
Set	N,K,V	A,K,V	A,V	A,N,V	A,N,K,V	A,N,K,V	A,N,K,V,X,I,O	A,N,V
Get	K	A,K,	A	A,N	A,N,K	A,K	A,N,K,X,I,O	A,N
Clear	K	A,K	A	A,N	A,N,K	A,K	A,N,K,X,I,O	A,N
Count	-	K,I	-	A,I	A,N,I,O	A,I	A,I	A,I
Member	K	A,K,I	A	A,N,I	A,N,K,I,O	A,K,I	A,K,I	A,N,I
Name	K,X	A,K,X,I	A,X	A,X,I	A,K,X,I,O	A,K,X,I	A,I,O	A,X,I
Key	X	A,X,I	A,X	A,N,X,I	A,N,X,I,O	A,X,I	A,X,I	A,N,X,I
Value	K,X	A,K,X,I	A,X	A,N,X,I	A,N,K,X,I,O	A,K,X,I	A,N,K,X,I,O	A,N,X,I
Index	K	A,K,I	A	A,N,I	A,N,K,I,O	A,K,I	A,K,I	A,N,I
Attribute	K,X	K,X,I	X	N,X,I	N,K,X,I,O	K,X,I	K,X,I	N,X,I
Purge	-	K,I	-	A,I	A,N,I,O	A,I	A,I	A,I
Sort	-	K,I	-	A,I	A,N,I,O	A,I	A,I	A,I
Retrieve	-	K,I	-	A,I	A,N,I,O	A,I	A,I	A,I

A = Attribute; N = Name; K = Key; V = Value; X = Index; I = Instance; O = Offset

Table 1. Avial Model Aspect Arguments

Adapter Restrictions

A typical Avial installation comes with five standard adapters that provide the model functionality describe above. These adapters are implemented using precisely the same software. However, to aid performance and computational load balancing, each adapter is configured different, placing restrictions on which model aspects are supported. Among an adapter's configuration parameters are the ability to enable or disable the main property and fact model aspect groupings, and whether or now the adapter supports data storage (i.e. file read/write). Table 2 contains a summary of these restrictions. While the

registries and objects adapters are restricted to just property-related aspects and fact-related aspects, the generals adapter is configured with no restrictions.

Adapter	Outlet	Properties	Facts	Data
Registries	<10>	X		
Objects	<11>		X	
Folders	<12>	X	X	
Files	<13>		X	X
Generals	<14>	X	X	X

Table 2. Avial Model Aspect Arguments

Entities

Permanently associated with every entity is an *entity unique identifier* (EUID). This identifier is guaranteed to be unique across an entire AvesTerra server constellation. An entity's EUID is provided by the AvesTerra framework at the moment an entity is created and can never be changed. Every EUID consists of 3 parts:

- Public identifier (PID)
- Host identifier (HID)
- Unique identifier (UID)

The PID of an entity identifies the public network where the AvesTerra server that manages the entity resides. Similarly, an entity's HID identifies the specific host within that public network. Lastly, the UID identifies the entity on the designated host. Internally, the PID and HID are encodings assigned by a server to enable routing through the AvesTerra peer-to-peer network. The three components together comprise a 128-bit identifier. For textual representation, an EUID is displayed as three decimal number groups separated by the '|' character enclosed in angle brackets in the form <*pid*|*hid*|*uid*>. Several of the AvesTerra tools allow the shorthand forms <*hid*|*uid*> and <*uid*> where the missing *pid* and *hid* fields are defaulted to that of the current server.

When an entity is created, the PID and HID are assigned by the server corresponding to the server's public and host network identifiers, respectively, and the UID portion of its EUID is assigned by a server in incremental order starting at 100001. When a server receives an entity request, it examines the PID and the HID to determine if it is responsible for managing that entity (i.e. it is the entity's "owner"). If the entity is managed by the local server, the request is dispatched for local fulfillment, otherwise the server uses the PID and HID to forward the request to the proper remote server destination, through the AvesTerra peer-to-peer network.

AvesTerra EUIDs are designed for efficient routing and are generally not very meaningful nor memorable to person. As a convention to help organize the AvesTerra entity identifier space, entities with UIDs in the range 0..100000 are reserved by each server for entity *redirection*. Entity redirection is a means of mapping a set of "common" (i.e. well-known) UIDs to actual (and generally not well-known) EUIDs. Every server maintains an entity redirection table for this purpose. When a server receives a reference to an entity with PID = 0, HID = 0, and UID < 1000001, the server looks up the entity's UID in the redirection table, replacing the reference with the EUID table entry. In this manner, a logical entity UID reference is redirected to an actual entity EUID reference. All common EUIDs are of the form <0|0|*uid*>. The Avial system assigns specific meaning to many of these reserved EUIDs. For example, Avial designates entity <0|0|1> as the "AvesTerra Registry", not unlike a parent directory in many conventional file systems.

While entity redirection is very helpful on a single server, the EUID of a common/well-known entity on one server may not be well-known on another server. Consequently, AvesTerra provides a means of common entity *forwarding*. That is, whenever a server receives a request for an entity with a EUID of the form <*pid*|*hid*|*uid*> where the UID is less than 100001 and the PID and HID are remote, the server will forward the entity request in the usual manner to the specified remote server. However, upon arrival the EUID will be reassigned <0|0|*uid*>. The remote server will then redirect this new common identifier to the entity's actual EUID on that server via its local redirection table. Using this scheme,

entity references on one server can be made to entities on another server without having to know the precise EUID in advance.

Lastly, entity <0|0|0> has special significance on every server. Specifically, <0|0|0> refers to the server itself. That is, the server recognizes entity <0|0|0> as a reference to its own instance of the running AvesTerra server software. In this manner, applications can examine various aspects of the running server, observe various configuration settings, and make certain configuration requests without the need for additional custom API operations. The server accomplishes this by organizing information about itself consistent with the Avial model. As a result, the same tools for accessing with conventional entities can be used for accessing the server, provided the requesting application has the appropriate authorization.

Authorizations

All entities in an AvesTerra knowledge space conform to an authorization model that is layered upon the AvesTerra server infrastructure. Nearly every AvesTerra operation requires an authorization. Specifically, every operation involving the creation or modification of an entity, or access to any of an entity's model information requires an authorization. An authorization is a digital token that the server uses to determine that the requestor of an operation has the appropriate permission. Depending upon how a server is configured, that token can default to a common token shared amongst multiple server clients, or it can be a unique token assigned and only known to a single client.

At the server API level, an authorization is a 128-bit identifier, compliant with Internet Engineering Task Force (IETF) Universal Universally Unique Identifier (UUID) standard RFC 4122. This token can either be selected by the client or randomly generated by various AvesTerra tools including the Avial package and the AvesTerra Visualization Utility (Avu). In text form, an authorization token is displayed in conformance with the RFC as a 36 character hyphenated hexadecimal (lower-case) sequence. For example:

`54a5a93a-c3cc-4ea6-8003-bb0a47bf941f`

For convenience during development, testing, and experimental deployments, several of the AvesTerra user interface tools allow a shorthand version when the first 80 bits are zero. That is, the authorization "00000000-0000-0000-0000-000123456789" can be abbreviated simply as "123456789". In addition, there are three authorization tokens that have a specific meaning. The *null_authorization* token is the default authorization, digitally containing all 0's. This token is used in situations where access permission is not required. Conversely, the *no_authorization* token digitally contains all 1's and is used to indicate that authorization is not permitted. This token mostly commonly occurs as the result of a request to view an entity's authorization without the proper authorization. Lastly, the *view_authorization* token (a single binary '1' in the last digit) can be used by entities to authorize read-only operations. That is, clients can examine the information content of an entity but not make any modifications. These special tokens are listed in Table 1. Note that several of the AvesTerra tools display the *no_authorization* token with the '*' character as shown in Table 1.

Authorization	**UUID**
null_authorization	`00000000-0000-0000-0000-000000000000`
view_authorization	`00000000-0000-0000-0000-000000000001`
no_authorization	`ffffffff-ffff-ffff-ffff-ffffffffffff` `********-****-****-****-************`

Table 1. Special Authorization Tokens

Every entity in the AvesTerra knowledge space has both an *authority* and an *authorizations* set. An entity's authority is the authorization that was used to create the entity. Once created, this authorization is needed to make server related changes to the entity (e.g. delete, create connections, event subscriptions, etc.). Without the proper authorization, these operation will fail, returning an

authorization exception and/or authorization error status. If an entity is created with a *null_authorization*, then there are no restrictions that prevent any application from making changes to the entity. While this may be common for development and testing applications, the *null_authorization* is clearly not advised for production environments. In addition to an entity's authority, the entity authorizations set is provided to allow other applications to use the entity without having to share the entity's authority. That is, applications will have the ability to execute the INVOKE and INQUIRE operations on the entity, but they will not have the ability to make configuration changes to the entity (e.g. connecting/disconnecting to/from adapters, attaching/detaching attributes, modifying reference counts, etc.). In addition, only an application that knows an entity's authority can make changes to the entity's authorization set. An application which knows an entity's authority can authorize "read-only" access to those that would otherwise have no authorized access to the entity. This can be accomplished by including the *view_authorization* token in the entity's authorization set.

When an AvesTerra server is initially configured, the server is assigned a top-level (i.e. "root") authority by the server's administrator. This can either be a randomly generated authorization, or one selected by the administrator. A server's authority is set via a request made to the server's entity <0|0|0>. Knowing this authority, an application can request the server to create and delete entities, make configuration changes, etc. To allow other applications to perform such server operations without having to share the server's authority, additional authorizations can be added to the server entity's authorization set. To perform such changes, and to change the server's authority itself, the server's current authority must be known and provided when a request is made. To view an entity's authorizations (including the entity's authority), either the entity's authority or the server's authority must be provided. Only a server's authority is authorized to view or make changes to a server's authority and/or its authorization set. If a server's authority becomes lost, there is no way to recover it via the AvesTerra API.

To help manage authorizations, the Avial Compartments Adapter is provided in every standard AvesTerra installation. This adapter is used by client applications to save and recover their authorizations at run-time, thus preventing them from having to be "hard-coded" into the application code, or entered manually be a user. Authorizations are organized using *compartments*, a collection of entities that share the same "need-to-know" access restrictions. Each compartment contains a unique authorization that generally is not shared. Rather, the compartment is assigned an authorization token that a server uses to map to the compartment's actual authorization. This is to allow compartment access to be readily revoked without having to assign new authorities to every entity contained within the compartment.

In order to access a compartment, a client application must first authenticate itself as a legitimate *identity* with the Authentication Adapter. An identity is an entity that is assigned to a unique party or software component set that requires access to compartments. The Authentication Adapter maintains a list of identities and the compartments, if any, that they have been authorized to access. To obtain a compartment's authorization token, a client must present the Authentication Adapter with a valid identity authorization token. To obtain this identity token, the client must present the Authentication Adapter with a valid identity name and password, analogous to a username/password. [Note that AvesTerra maintains no notion of a "login session", though higher-level applications may choose to implement such a mechanism.] Client applications can only obtain their identity's authorization token only if the identity name and password they provide are correct. In addition, the identity has to have previously been validated (once at a minimum) using a signature verification process. To accomplish

this, a user associated the identity has to have first generated a unique public/private key pair (the Avu program contains a command to do this). The public key portion must be installed on the server by its administrator. This allows the Compartment Adapter to exchange a randomly generated code with the client and confirm using the identity's public key that the client can properly encrypt the random code. Once a validated identity is authenticated, the client can then receive the identity's authorization token. With this token, the client can then present this token to the Authentication Adapter to request the authorization token for the compartment. The compartment token will only be returned by the Authentication Adapter if the identity token is properly authorized for the compartment.

Note that the server has no actual knowledge of compartments nor identities. These are constructs created and managed solely by the Authentication Adapter. The server knows only of entity authorities and entity authorizations. Thus, multiple authentication schemes are possible, depending upon specific knowledge space access control requirements. The server does, however, perform automatic token to authorization mapping so that an entity's actual authority/authorization need not be visible to a client. In general, the Authentication Adapter on each server runs as a protected application under the strict control of the server's administrator. In some AvesTerra environments where highest levels of privacy assurance are required, a system may not have an administrator. That is, the Authentication Adapter is entirely self-contained and self-configured (e.g. the ATra system "blackbox").

In summary, to create/delete entities on a server and to connect/disconnect these entities to adapters, either the server's authority must be known, or an authorization that was added to the server's authorization set must be known. In order to access any of an entity's methods or attributes (i.e. Avial functions built upon the API INVOKE and INQUIRE operations), either the authority of the entity must be known, or an authorization that was added to the entity's authorization set must be known. The Compartment Adapter helps manage these on clients' behalf using public/private key signature verification and passwords.