Remote Administration Daemon Developer Guide

Part No: E35864–01
October 2012

ORACLE

121010@25097

Contents

Preface

The Oracle Solaris operating system can be administered or configured remotely. Applications that allow you to remotely administer or configure a system, require programmatic access. This guide provides information on how to use the remote administration daemon to provide programmatic access to the administration and configuration functionality of the Oracle Solaris OS.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Description	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.

TABLE P–1 Typographic Conventions *(Continued)*

Typeface	Description	Example
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows UNIX system prompts and superuser prompts for shells that are included in the Oracle Solaris OS. In command examples, the shell prompt indicates whether the command should be executed by a regular user or a user with privileges.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

◆ ◆ ◆ CHAPTER 1

Introduction

The Remote Administration Daemon, commonly referred to by its acronym and command name, `rad`, is a standard system service that offers secure, remote administrative access to an Oracle Solaris system. This book is intended for developers who want to use `rad` to create administrative interfaces and for developers looking to consume interfaces published using `rad` by others. It introduces `rad` and its core concepts, explains the process of developing `rad` extensions and consumers, and includes a reference for the `rad` module API and client libraries.

Remote Administration Daemon

The Oracle Solaris operating system is a set of cooperating components, and correspondingly administering Oracle Solaris is the act of manipulating those components in a variety of ways. The traditional solution consisted of locally applying `$EDITOR` to text files. More modern approaches include manipulating system components locally using a CLI or an interactive UI, remotely with a browser or client, *en masse* with an enterprise-scale provisioning tool, or automatically by policy logic employed by any of these methods. All of these methods require programmatic access to configuration. The Remote Administration Daemon is the central point where system developers can expose their components for configuration or administration, and where the various programmatic consumers can go to perform those activities.

To fully support consumers written in a variety of languages, consumers running without privilege, and consumers running remotely, `rad` employs a client/server design. `rad` itself acts as a server that services remote procedure calls. `rad` consumers are the clients. The protocol `rad` speaks is efficient and easy to implement, which makes it simple to bring support for *all* administrative tasks exposed via `rad` to a new language.

By providing a procedure call interface, `rad` enables non-privileged local consumers to perform actions on behalf of their users that require elevated privilege, without needing to resort to a CLI-based implementation. Finally, by establishing a stream protocol, these same benefits can be extended to consumers on any machine or device over a variety of secure transport options.

rad differs from traditional RPC in a number of ways:

- Procedure calls are made against server objects in a browsable, structured namespace. This process permits a more natural evolution of content than is afforded by the central allocation of program numbers.
- These procedure calls need not be synchronous. Subject to the protocol in use, a client may have multiple simultaneous outstanding requests.
- The interfaces exported by the server objects are fully inspectable. This facilitates interactive use, rich debugging environments, and clients using dynamically-typed languages such as Python.
- In addition to defining procedure calls, rad objects can define properties and asynchronous event sources. Though the former provides more of a semantic than a functional improvement, the latter is a powerful tool for efficiently observing changes to the system being managed.

Note –

- rad supports alternate protocols without needing to update its content, which provides even greater flexibility.
- rad's native protocol fully supports asynchronous procedure calls once the client has authenticated. An alternate protocol, e.g. one based on XML-RPC, might not support asynchronous calls due to limitations of the underlying technology.

Features Overview

The main functionality offered by rad is as follows:

- **Essentials**
  - Managed and configured by two SMF services, svc:/system/rad:local and svc:/system/rad:remote
  - Structured, browsable namespace.
  - Inspectable, typed, versioned interfaces.
  - Asynchronous event sources.
  - XML-based IDL ADR supports formally defining APIs. The IDL compiler radadrgen generates code and analyzes compatibility of API changes.
- **Security**
  - Full PAM conversation support including use of pam_setcred(3PAM) to set the audit context.
  - Implicit authentication using getpeerucred(3C) when possible.
  - No non-local network connectivity by default. Preconfigured to use TLS.

- Most operations automatically delegated to lesser-privileged processes.
- Defines two authorizations (`solaris.smf.manage.rad` and `solaris.smf.value.rad`) and two Rights Profiles (`rad` Management and `rad` Configuration) to provide fine-grained separation of powers for managing and configuring the `rad` SMF services.

  rad authorizations

  `solaris.smf.manage.rad` — Grants the authorization to enable, disable, or restart the `rad` SMF services.

  `solaris.smf.value.rad` — Grants the authorization to change `rad` SMF services' property values.

  rad rights profiles

  `rad` Management — Includes the `solaris.smf.manage.rad` authorization.

  `rad` Configuration — Includes the `solaris.smf.value.rad` authorization.
- Generates `AUE_rad_login`, `AUE_logout`, `AUE_role_login`, `AUE_role_logout`, and `AUE_passwd` audit events.

- **Connectivity**
  - Local access via `AF_UNIX` sockets.
  - Remote access via TCP sockets.
  - Secure remote access via TLS sockets.
  - Captive execution with access through a pipe.
  - Connection points are completely configurable at the command line or via SMF.
- **Client support**
  - A JMX connector exposes `rad` interfaces as Open MBeans to Java clients.
  - `radadrgen` can auto-generate MXBean interfaces for use by JMX consumers. Can also generate basic concrete subclasses.
  - A plain Java client permits explicit access to server objects from non-JMX environments.
  - A native C library offers explicit access to server objects.
- **Extension**
  - A public native C module interface supports addition of third-party content.
  - `radadrgen` can generate server-side type definitions and stubs from IDL input.
  - A native execution system can automatically run modules with authenticated user's privilege and audit context, simplifying authentication and auditing.
  - Private module interfaces permit defining new transports.

◆ ◆ ◆ CHAPTER 2

Concepts

The concepts that are fundamental to `rad` are *interfaces*, objects that implement those interfaces, and the namespace in which those objects can be found and operated upon. This chapter discusses the following concepts.

This chapter discusses the following concepts that are fundamental to `rad`.

Interface

An interface defines how a `rad` client can interact with an object. An object implements an interface, providing a concrete behavior to be invoked when a client makes a request.

The primary purpose of `rad` is to consistently expose the various pieces of the system for administration. Not all subsystems are alike, however: each has a data and state model tuned to the problems they are solving. Although there are major benefits to using a common model across components when possible, uniformity comes with trade-offs. The increased inefficiency and client complexity, and risk of decreased developer adoption, often warrant using an interface designed for problem at hand.

An interface is a formal definition of how a client may interact with a `rad` server object. An interface may be shared amongst several objects, for example, when maintaining a degree of uniformity is possible and useful, or may be implemented by only one. A `rad` interface is analogous to an interface or pure abstract class in an object oriented programming language. In the case of `rad`, an interface consists of a name, the set of features a client may interact with, optionally a set of derived types referenced by the features, and a version. The features supported include:

- Methods, which are procedure calls made in the context of a specific object

- Properties, which are functionally equivalent to methods but bear different semantics
- Asynchronous event sources

Name

Each interface has a name. This name is used by the toolchain to construct identifier names when generating code, and is returned by the server along with the rest of the interface definition when an object is examined by a client. There is no global namespace for interfaces, however. A client is expected either to know which objects implement which interfaces (typical consumer) or to query `rad` for the object's full interface definition (debugger or interactive tools).

Derived Types

Three classes of derived types may be defined for use by features: structures, unions, and enumerations. Each type defined must be uniquely named. As with interfaces, there is no global type namespace. The types defined in an API are available only to the features defined in that API.

Features

The only thing all three feature types — methods, attributes, and events — have in common is that they are named. All three feature types' names exist in the same namespace and must therefore be unique. You can not have both a method and an attribute called “foo.” This exclusion avoids the majority of conflicts that could arise when trying to naturally map these interface features to a client environment.

Note – Enforcing a common namespace for interface features isn't always enough. Some language environments place additional constraints on naming. For instance, a JMX client using MXBean proxies will see an interface with synthetic methods of the form `getXXX`() or `setXXX`() for accessing attribute `XXX` that must coexist with other method names. Explicitly defining methods with those names will cause a conflict.

Methods

A method is a procedure call made in the context of the object it is called on. In addition to a name, a method may define a return type, can define zero or more arguments, and may declare that it returns an error, optionally with an error return type.

If a method does not define a return type, it returns no value. It is effectively of type void. If a method defines a return type and that type is permitted to be nullable, the return value may be defined to be nullable.

Each method argument has a name and a type. If any argument's type is permitted to be nullable, that argument may be defined to be nullable.

If a method does not declare that it returns an error, it theoretically cannot fail. However, because the connection to `rad` could be broken either due to a network problem or a catastrophic failure in `rad` itself, all method calls can fail with an I/O error. If a method declares that it returns an error but does not specify a type, the method may fail due to API-specific reasons. Clients will be able to distinguish this failure type from I/O failures.

Finally, if a method also defines an error return type, data of that type may be provided to the client in the case where the API-specific failure occurs. Error payloads are implicitly optional, and must therefore be of a type that is permitted to be nullable.

Note – Methods names may not be overloaded.

Attributes

An attribute is metaphorically a property of the object. Attributes have the following characteristics:

- A name
- A type
- A definition as read-only, read-write, or write-only
- Like a method may declare that accessing the attribute returns an error, optionally with an a error return type

Reading a read-only or read-write attribute returns the value of that attribute. Writing a write-only or read-write attribute sets the value of that attribute. Reading a write-only attribute or writing a read-only attribute is invalid. Clients may treat attempts to write to a read-only attribute as a write to an attribute that does not exist. Likewise, attempts to read from a write-only attribute may be treated as an attempt to read from an attribute that does not exist.

If an attribute's type is permitted to be nullable, its value may be defined to be nullable.

An attribute may optionally declare that it returns an error, with the same semantics as declaring (or not declaring) an error for a method. Unlike a method, an attribute may have different error declarations for reading the attribute and writing the attribute.

Attribute names may not be overloaded. Defining a read-only attribute and a write-only attribute with the same name is not valid.

Given methods, attributes are arguably a superfluous interface feature. Writing an attribute of type X can be implemented with a method that takes one argument of type X and returns nothing, and reading an attribute of type X can be implemented with a method that takes no arguments and returns a value of type X. Attributes are included because they have slightly different semantics.

In particular, an explicit attribute mechanism has the following characteristics:

- Enforces symmetric access for reading and writing read-write attributes.
- Can be easily and automatically translated to a form natural to the client language-environment.
- Communicates more about the nature of the interaction. Reading an attribute ideally should not affect system state. The value written to a read-write attribute should be the value returned on subsequent reads unless an intervening change to the system effectively "writes" a new value.

Events

An event is an asynchronous notification generated by `rad` and consumed by clients. A client may subscribe to events by name to register interest in them. The subscription is performed on an object which implements an interface. In addition to a name, each event has a type.

Events have the following characteristics:

- Sequential.
- Volatile
- Guaranteed

A client can rely on sequential delivery of events from a server as long as the connection to the server is maintained. If the connection fails, then events will be lost. On reconnection, a client must resubscribe to resume the flow of events.

Once a client has subscribed to an event, event notifications will be received until the client unsubscribes from the event.

On receipt of a subscribed event, a client receives a payload of the defined type.

Versioning

`rad` interfaces are versioned for the following reasons:

- APIs change over time.
- A change to an API might be incompatible with existing consumers.
- A change might be compatible with existing consumers but new consumers might not be able to use the API that was in place before the change occurred.
- Some features represent committed interfaces whose compatibility is paramount, but others are private interfaces that are changed only in lockstep with the software that uses them.

Numbering

The first issue is measuring the compatibility of a change. `rad` uses a simple `major.minor` versioning scheme. When a compatible change to an interface is made, its minor version number is incremented. When an incompatible change is made, its major version number is incremented and its minor version number is reset to 0.

In other words, an implementation of an interface that claims to be version X.Y (where X is the major version and Y is the minor version) must support any client expecting version X.Z, where Z <= Y.

The following interface changes are considered compatible:

- Adding a new event
- Adding a new method
- Adding a new attribute
- Expanding the access supported by an attribute, for example, from read-only to read-write
- A change from nullable to non-nullable for a method return value or readable property, that is, decreasing the range of a feature
- A change from non-nullable to nullable for a method argument or writable property, that is, increasing the domain of a feature

The following interface changes are considered incompatible:

- Removing an event
- Removing a method
- Removing an attribute
- Changing the type of an attribute, method, or event
- Changing a type definition referenced by an attribute, method, or event
- Decreasing the access supported by an attribute, for example, from read-write to read-only
- Adding or removing method arguments
- A change from non-nullable to nullable for a method return value or readable property, that is, increasing the range of a feature
- A change from nullable to non-nullable for a method argument or writable property, that is, decreasing the domain of a feature

---

Note – An interface is more than just a set of methods, attributes, and events. Associated with those features are well-defined behaviors. If those behaviors change, even if the structure of the interface remains the same, a change to the version number might be required.

Commitment

To solve the problem of different features being intended for different consumers, rad defines three commitment levels: private, uncommitted, and committed. Each method, attribute, and event in an interface defines its commitment level independently. The interface is assigned a separate version number, per commitment level.

Each commitment level is considered a superset of the next more-committed level. For example, "private" is a superset of "uncommitted." Therefore, when an uncommitted (or committed) interface changes, the private version number needs to be changed as well. By having separate version numbers instead of just adding more dots to the existing one, private/uncommitted consumers are not broken by compatible changes to uncommitted/committed interfaces.

When a feature changes commitment level, it is treated as if the feature was removed from the old commitment level and added to the new one. If a feature becomes less committed, then that implies an incompatible change for the every commitment level that no longer includes that feature but no change for every commitment level that still includes the feature due to the implicit nesting of commitment levels. If a feature becomes more committed, then that implies a compatible change for each commitment level that gained the feature and no change for each commitment level that had it before.

The simple case of an API containing interfaces of only a single commitment level reduces to traditional commitment-agnostic major/minor versioning.

Clients and Versioning

A rad client can ask for interface version information if the protocol in use does not automatically provide it. The client decides what to do with this information. It can expose that information directly to interface consumers, or it can provide APIs that encapsulate verifying that the version the client expects is the version the server is providing.

rad Namespace

The namespace acts as rad's gatekeeper, associating a name with each object, dispatching requests to the proper object, and providing meta-operations that enable the client make queries about what objects are available and what interfaces they implement.

A rad server may provide access to several objects that in turn expose a variety of different components of the system or even third-party software. A client merely knowing that interfaces exist, or even that a specific interface exists, is not sufficient. A simple, special-purpose client needs some way to identify the object implementing the correct interface with the correct behavior, and an adaptive or general-purpose client needs some way to determine what functionality the rad server has made available to it.

rad organizes the server objects it exposes in a namespace. Much like files in a file system, objects in the rad namespace have names that enable clients to identify them, can be acted upon or inspected using that name, and can be discovered by browsing the namespace. Depending on the point of view, the namespace either is the place one goes to find objects or the intermediary that sits between the client and the objects it accesses. Either way, it is central to interactions between a client and the rad server.

Naming

Unlike a file system, which is a hierarchical arrangement of simple filenames, rad adopts the model used by JMX and maintains a flat namespace of structured names. An object's name consists of a mandatory reverse-dotted domain combined with a non-empty set of key-value pairs. There aren't any restrictions on what a key or value can contain, which can make representing them as a string difficult. For the sake of simplicity, this document does not attempt to establish a canonical form, but uses a form similar to the serialized form of JMX ObjectNames:

```
domain:key1=value1[,key2=value2[,...]]
```

See "Naming Guidelines" on page 70 for guidance on object naming. For more information about JMX, see the Java Management Extensions (JMX) technology home page (http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html) .

Equality

Two names are considered equal if they have the same domain and the same set of keys, and each key has been assigned the same value.

Patterns

Some situations call for referring to groups of objects. In these contexts, a name can be used as a pattern. Another object name matches a pattern if all components present in the pattern are present and equal in the name. In these contexts, name used as patterns are permitted to have an unspecified domain or an empty set of key-value pairs. For example, the pattern :product=fruit (that is, no domain, and only the key "product" with the value "fruit" specified) would match the names `grocery.bob:product=fruit,type=banana` and `grocery.jim:product=fruit,type=apple`, but not `grocery.bob:product=animal,type=fish` or `grocery.bob:person=shelver`.

Operations

Just as names are essential to a client's interaction with the rad server, so are the actions the client performs. While the exact nature of the requests a client performs are a function of the protocol used and might be hidden from the developer by the client implementation itself, defining the set of abstract operations supported by the rad namespace is still useful.

LIST	Requests that the objects of the namespace be enumerated. Returns a list of object names.
DESCRIBE	Requests a description of a specific object by name. Returns an interface description.
INVOKE	Invokes a method on the specified object with the specified arguments. Returns the result of that method invocation, if any.
GET	Reads the value of an attribute of the specified object.
SET	Writes a value to an attribute of the specified object.
SUBSCRIBE	Subscribes to the specified event source of the specified object.
UNSUBSCRIBE	Unsubscribes from the specified event source of the specified object.

LIST and DESCRIBE are analogous to the VFS operations READDIR and LOOKUP, respectively. As is the case with VFS, it might be possible to call DESCRIBE on objects that are not enumerated by LIST. The usual reason for this arrangement applies here as well: enumerating the available objects might be prohibitively expensive, either for algorithmic reasons or due to their sheer quantity. The bandwidth required for a large response (and the corresponding latency induced in every client application) can also be a consideration. Additionally, the server objects that rad provides might represent underlying objects that themselves subject their consumers (of which rad is one) to such a restriction.

This type of arrangement also enables renaming of objects. The server can respond to requests against an old name to maintain compatibility with legacy clients without needing to broadcast the existence of those old names.

Note – These operations represent the most basic set of operations related to discussing communication with a rad server. For efficiency, the rad protocol further divides DESCRIBE into separate LOOKUP and DEFINE operations.

Data Typing

All data returned submitted to or obtained from rad APIs adheres to a strong typing system similar to that defined by XDR. For more information about XDR, see the XDR (http://tools.ietf.org/rfc/rfc4506.txt) standard. This makes it simpler to define interfaces that have precise semantics, and makes server extensions (which are written in C) easier to develop. Of course, the rigidity of the typing exposed to an API's consumer is primarily a function of the client language and implementation.

Base Types

rad supports the following base types:

boolean	A boolean value (true or false).
integer	A 32-bit signed integer value.
uinteger	A 32-bit unsigned integer value.
long	A 64-bit signed integer value.
ulong	A 64-bit unsigned integer value.
float	A 32-bit floating-point value.
double	A 64-bit floating-point value.
string	A UTF-8 string.
opaque	Raw binary data.
secret	An 8-bit clean “character” array. The encoding is defined by the interface using the type. Client/server implementations may take additional steps, for example, zeroing buffers after use, to protect the contents of secret data.
time	An absolute UTC time value.
name	The name of an object in the rad namespace.

Derived Types

In addition to the base types, rad supports several derived types.

An enumeration is a set of user-defined tokens. Like C enumerations, rad enumerations may have specific integer values associated with them. Unlike C enumerations, rad enumerations and integers are not interchangeable. Among other things, this aspect means that an

enumeration data value may not take on values outside those defined by the enumeration, which precludes the common but questionable practice of using enumerated types for bitfield values.

`rad` enumerations support designating an optional "fallback" value. A fallback value enables an enumeration to change without breaking compatibility between consumers and implementors of an API using different versions of the enumeration. When supported by the programming environment in use, the connection between the consumer and the implementor will automatically map unrecognized enumeration data values to the fallback value.

Caution – Fallback values are indispensible in cases where the set of possible enumeration must change over time. However, any change to an enumeration with a fallback value is considered to be a compatible change, forfeiting some of the benefits offered by `rad`'s versioning scheme. Excessive use of fallback values will unnecessarily complicate the use and maintenance of a `rad` API.

A discriminated union is a data type that can be used to store polymorphic data, with typesafe access to the data. Like XDR, discriminated union's are composed of a set of "arms" and a "discriminant" that selects an "arm." The discriminant may be a boolean or an enumeration type. A default arm may be specified for unions with an enumerated discriminator. When no arm is defined for a specific discriminant value and no default arm is defined, the arm for that value is void.

An array is an ordered list of data items of a fixed type. Arrays do not have a predefined size.

A structure is a record consisting of a fixed set of typed, uniquely named fields. A field's type may be a base type or derived type, or even another structure type.

Derived types offer almost unlimited flexibility. However, one important constraint imposed on derived types is that recursive type references are prohibited. Thus, complex self-referencing data types, for example, linked lists or trees, must be communicated after being mapped into simpler forms.

Optional Data

In some situations, data may be declared as nullable. Nullable data can take on a "non-value", for example, `NULL` in C, or null in Java. Inversely, non-nullable data cannot be `NULL`. Only data of type opaque, string, secret, array, union or structure may be declared nullable. Additionally, only structure fields and certain API types can be nullable. Specifically, array data cannot be nullable because the array type is actually more like a list than an array.

◆ ◆ ◆ CHAPTER 3

Abstract Data Representation

The data model used by `rad` is known as the Abstract Data Representation (ADR). ADR defines a formal IDL for describing types and interfaces supplies a toolchain for operating on that IDLand provides libraries used by `rad`, its extension modules, and its clients.

ADR Interface Description Language

The APIs used by `rad` are defined using an XML-based IDL. The normative schema for this language can be found in `/usr/share/lib/xml/rng/adr.rng.1`. The namespace name is `http://xmlns.oracle.com/radadr`.

Overview

The top-level element in an ADR definition document is an `api`. The `api` element has one attribute, `name`, which is used to name the output files. The element contains one or more derived type or interface definitions. Because there is no requirement that an interface use derived types, there is no requirement that any derived types be specified in an API document. To enable consumers to use the data typing defined by ADR for non-interface purposes, there is no requirement that an interface is defined either. However, note that either a derived type or an interface must be defined.

Three derived types are available for definition and use by interfaces: a structured type that can be defined with a `struct` element, an enumeration type that can be defined with an `enum` element, and a union type that can be defined with a `union` element. Interfaces are defined using `interface` elements. The derived types defined in an API document are available for use by all interfaces defined in that document.

EXAMPLE 3–1 Skeleton API document

```
<api xmlns="http://xmlns.oracle.com/radadr" name="example">
        <struct>...</struct>
        <struct>...</struct>
```

EXAMPLE 3–1 Skeleton API document *(Continued)*

```
        <enum>...</enum>
        <union>...</union>
        <interface>...</interface>
        <interface>...</interface>
</api>
```

Enumeration Definitions

The `enum` element has a single mandatory attribute, `name`. The `name` is used when referring to the enumeration from other derived type or interface definitions. An `enum` contains one or more `value` elements, one for each user-defined enumerated value. A `value` element has a mandatory `name` attribute that gives the enumerated value a symbolic name. The symbolic name isn't used elsewhere in the API definition, only in the server and various client environments. How the symbolic name is exposed in those environments is environment-dependent. An environment offering an explicit interface to `rad` should provide an interface that accepts the exact string values defined by the `value` elements' `name` attributes.

An `enum` also contains zero or one `fallback` elements, indicating that the enumeration has a fallback value. The `fallback` element must appear after all `value` elements when present. Like `value` elements, a `fallback` element has a `name` attribute.

Some language environments support associating scalar values with enumerated type values, for example C. To provide richer support for these environments, ADR supports this concept as well. By default, an enumerated value has an associated scalar value 1 greater than the preceding enumerated value's associated scalar value. The first enumerated value is assigned a scalar value of 0. Any enumerated value element may override this policy by defining a `value` with the desired value. A `value` attribute must not specify a scalar value already assigned, implicitly or explicitly, to an earlier value in the enumeration.

`value` elements contain no other elements.

EXAMPLE 3–2 Enumeration Definition

```
<enum name="Colors">
<value name="RED" /> <!-- scalar value: 0 -->
<value name="ORANGE" /> <!-- scalar value: 1 -->
<value name="YELLOW" /> <!-- scalar value: 2 -->
<value name="GREEN" /> <!-- scalar value: 3 -->
<value name="BLUE" /> <!-- scalar value: 4 -->
<value name="VIOLET" value="6" /> <!-- indigo was EOLed -->
<fallback name="UNKNOWN" /> <!-- for compatibility -->
</enum>
```

Structure Definitions

Like the `enum` element, the `struct` element has a single mandatory attribute, `name`. The name is used when referring to the structure from other derived type or interface definitions. A `struct` contains one or more `field` elements, one for each field of the structure. A `field` element has a mandatory `name` attribute that gives the field a symbolic name. The symbolic name isn't used elsewhere in the API definition, only in the server and various client environments. In addition to a name, each field must specify a type.

You can define the type of a field in multiple ways. If a field is a plain base type, that type is defined with a `type` attribute. If a field is a derived type defined elsewhere in the API document, that type is defined with a `typeref` attribute. If a field is an array of some type (base or derived), that type is defined with a nested `list` element. The type of the array is defined in the same fashion as the type of the field: either with a `type` attribute, a `typeref` attribute, or another nested `list` element.

A field's value may be declared nullable by setting the `field` element's `nullable` attribute to true.

Note – Structure fields, methods return values, method arguments, attributes, error return values, and events all have types, and in the IDL, use identical mechanisms for defining those types.

EXAMPLE 3–3 struct Definition

```
<struct name="Name">
        <field name="familyName" type="string" />
        <field name="givenNames">
                <list type="string" />
        </field>
</struct>

<struct name="Person">
        <field name="name" typeref="Name" />
        <field name="title" type="string" nullable="true" />
        <field name="shoeSize" type="int" />
</struct>
```

Union Definitions

The union element has the structure shown in the following example:

EXAMPLE 3–4 union Definition

```
<!-- For booleans -->

<union name="boolunion" type="boolean">
    <arm value="true" type="int" />
```

EXAMPLE 3–4 union Definition *(Continued)*

```
    <arm value="false" typeref="banana" />
</union>

<!-- For enumerations -->

<enum name="enumname">
    <value name="eval1" />
    <value name="eval2" />
    ...
</enum>

<union name="enumunion" typeref="enumname">
    <arm value="eval1">
        <list type="string" />
    <arm value="eval2" typeref="boolunion">
    <default type="int" />
</union>
```

The type of the union discriminator is designated by the `type` attribute, or `typeref` attribute, and may only be a boolean or enumerated type. The names of `union`s are in the same namespace as enumerated types and structures, and must be unique. `arm`s are identified with their associated `value`'s name. One default `arm` may be specified for `union`s with an enumerated discriminator. When no `arm` is defined for a value and no default `arm` is defined, the `arm` for that value is void. The default `arm` is optional.

Interface Definitions

An interface definition has a name, zero or more version specifications, and one or more attributes, methods, or events. An interface's name is defined with the `interface` element's mandatory `name` attribute. This name is used when referring to the inherited interface from other interface definitions, as well as in the server and various client environments. The other characteristics of an interface are defined using child elements of the `interface` element.

Version

An interface may define one version for each commitment level. You do not have to define a version for every commitment level, but one should be specified for every commitment level assigned to the features defined in the interface, that is, the features defined in the enclosing `interface` element.

A version is defined using a `version` element. The commitment level being versioned is defined by the mandatory `stability` attribute, which takes a value of committed, uncommitted, or private. The major and minor version numbers are non-negative integers, defined separately by the mandatory `major` and mandatory `minor` attributes.

EXAMPLE 3–5 Version Definition

```
<version stability="committed" major="2" minor="1" />
```

Methods

Each method in an interface is defined by a `method` element. The name of a method is defined by this element's mandatory `name` attribute. The other properties of a method are defined by child elements of the `method`.

If a method has a return value, it is defined using a single `result` element. The type of the return value is specified in the same way the type is specified for a structure field. If no `result` element is present, the method has no return value.

If a method can fail for an API-specific reason, it is defined using a single `error` element. The type of an error is specified the same way the type is specified for a structure field. Unlike a structure field, an error need not specify a type — such a situation is indicated by an `error` element with no attributes or child elements. If no `error` element is present, the method will only fail if there is a connectivity problem between the client and the server.

A method's arguments are defined, in order, with zero or more `argument` elements. Each `argument` element has a mandatory `name` attribute. The type of an argument is specified in the same way the type is specified for a structure field.

EXAMPLE 3–6 Method Definition

```
<struct name="Meal">...</struct>
<struct name="Ingredient">...</struct>

<method name="cook">
        <result typeref="Meal" />
        <error />
        <argument type="string" name="name" nullable="true" />
        <argument name="ingredients">
                <list typeref="Ingredient" />
        </argument>
</method>
```

Attributes

Each attribute in an interface is defined by a `property` element. The name of an attribute is defined by this element's mandatory `name` attribute. The types of access permitted are defined by the mandatory `access` attribute, which takes a value of `ro`, `wo`, or `rw`, corresponding to read-only access, write-only access, or read-write access, respectively.

The type of an attribute is specified in the same way the type is specified for a structure field.

If access to an attribute can fail for an API-specific reason, it is defined using one or more `error` elements. An `error` element in a `property` may specify a `for` attribute, which takes a value of `ro`, `wo`, or `rw`, corresponding to the types of access the error return definition applies to. An

`error` element with no `for` attribute is equivalent to one with a `for` attribute set to the access level defined on the `property`. Two error elements may not specify overlapping access types. For example, on a read-write property it is invalid for one `error` to have no `for` attribute (implying `rw`) and one to have a `for` attribute of `wo` — they both specify an error for writing.

The type of an error is specified the same way the type is specified for a method. It is identical to defining the type of a structure, with the exception that a type need not be defined.

EXAMPLE 3–7 Attribute Definition

```
<struct name="PrivilegeError">...</struct>

<property name="VIPList" access="rw">
        <list type="string" />
        <error for="wo" typeref="PrivilegeError" />
        <!-- Reads cannot fail -->
</property>
```

Events

Each event in an interface is defined by a `event` element. The name of an event is defined by this element's mandatory `name` attribute. The type of an event is specified in the same way the type is specified for a structure field.

EXAMPLE 3–8 Event Definition

```
<struct name="TremorInfo">...</struct>

<event name="earthquakes" typeref="TremorInfo" />
```

Pragmas

Occasionally you need to provide auxiliary information to a specific ADR consumer. The ADR IDL pragma mechanism lets an API creator specify that information in-line.

A pragma is specified with a `pragma` child element of the `api`. A pragma has no child elements, and three mandatory attributes. The domain attribute indicates which consumer the pragma is intended for, and a `name` and a `value` specify a consumer-specific name/value pair.

The only supported pragma specifies a package for generated Java classes. The domain of this pragma is `java` and its name is package.

EXAMPLE 3–9 Pragma Definition

```
<pragma domain="java" name="package" value="com.example" />
```

Example

EXAMPLE 3–10 Complete API Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<api xmlns="http://xmlns.oracle.com/radadr" name="example">

        <pragma domain="java" name="package" value="com.example" />

        <struct name="StringInfo">
                <field type="integer" name="length" />
                <field name="substrings">
                        <list type="string" />
                </field>
        </struct>

        <struct name="SqrtError">
                <field type="float" name="real" />
                <field type="float" name="imaginary" />
        </struct>

        <enum name="Mood">
                <value name="IRREVERENT" />
                <value name="MAUDLIN" />
        </enum>

        <struct name="MoodStatus">
                <field typeref="Mood" name="mood" />
                <field type="boolean" name="changed" />
        </struct>

        <interface name="GrabBag">
                <version major="1" minor="2" stability="private" />

                <method name="sqrt">
                        <result type="integer" />
                        <error typeref="SqrtError" />
                        <argument type="integer" name="x" />
                </method>

                <method name="parseString">
                        <result typeref="StringInfo" nullable="true" />
                        <argument type="string" name="str" nullable="true" />
                </method>

                <property typeref="Mood" name="mood" access="rw">
                        <error for="wo" />
                </property>

                <event typeref="MoodStatus" name="moodswings" />
        </interface>
</api>
```

radadrgen

`radadrgen` is the ADR IDL processing tool. Its primary purpose is to generate API-specific language bindings for the `rad` server and various client environments. It can also generate documentation, and can audit changes to interfaces for consistency with their versions. See the `radadrgen`(1) man page for details on all its options.

Code Generation

Generated code has advantages and disadvantages. On the one hand, it can provide an interface to a foreign system that acts like a natural part of the consumer's programming environment. On the other, it can introduce a complex maintenance burden if the generated code is flawed, constrains how its consumers are implemented (for example, if the generated code is unsafe to use in a threaded environment), or otherwise needs modifications before it can be used in the consumer's environment.

Because the `rad` server has complete knowledge of the interfaces and data types used, and that knowledge is explicitly shared with clients, any client can be written to manipulate those data types and communicate with the server without the need for generated code. However, the convenience of language-native interfaces increases developer productivity, their nature improves interoperability, and the ability for the compiler to perform additional type checking makes the resulting product more robust. For these reasons, `rad` supports generating code for the C and Java environments.

`radadrgen` needs to generate code for two contexts. The first is a generic, definitions-only "client" context where only the code needed to manipulate data types and interfaces is created. The second is a `rad`-specific context that, in addition to the definitions generated in the generic context, generates server definitions that include references to functions that form the implementation of the interfaces.

◆ ◆ ◆ CHAPTER 4

libadr

The library `libadr` provides structure definitions and subroutines essential to C programs using ADR. Code generated by `radadrgen` requires this library, and `rad` itself is based on it. `libadr` contains three major areas of functionality: data management, API management, and object name operations.

`libadr` is delivered in the `system/management/rad` package. C programs can link with it by specifying -ladr on the compile or link line.

Data Management

Consumers of the ADR data management routines should include the `rad/adr.h` header file:

```
#include <rad/adr.h>
```

This file contains definitions for the two fundamental data management types, `adr_type_t` and `adr_data_t`, as well as prototypes for data allocation, access, and validation routines.

adr_type_t Type

Each data type is represented by a `adr_type_t` type, whether it is just a base type or a complex of nested structures and arrays. The `adr_type_t` contains all the information necessary to understand the structure of the type. `libadr` provides statically-allocated singleton of `adr_type_t` type for the base types. These singleton types are more than a convenience: they must be used when referencing the base types.

The base types and their corresponding array types are listed in the following table.

TABLE 4–1 Base and Array Types

ADR type	C adr_type_t	C array adr_type_t
string	adr_t_string	adr_t_array_string
integer	adr_t_integer	adr_t_array_integer
uinteger	adr_t_uinteger	adr_t_array_uinteger
long	adr_t_long	adr_t_array_long
ulong	adr_t_ulong	adr_t_array_ulong
time	adr_t_time	adr_t_array_time
name	adr_t_name	adr_t_array_name
boolean	adr_t_boolean	adr_t_array_boolean
opaque	adr_t_opaque	adr_t_array_opaque
secret	adr_t_secret	adr_t_array_secret
float	adr_t_float	adr_t_array_float
double	adr_t_double	adr_t_array_double

The `adr_type_t` for a derived type should also be unique, but obviously they cannot be defined by libadr. Although technically `adr_type_t` could be dynamically allocated, at the moment, the only supported way of defining a `adr_type_t` is to generate a definition using the ADR IDL and `radadrgen`.

adr_data_t Type

The most frequently used type defined by `rad/adr.h` is `adr_data_t`. A `adr_data_t` object represents a unit of typed data. It could be of a base type, such as an integer ("1") or string ("banana"), or of a derived type like a structure or an array. Each `adr_data_t` maintains a pointer to its `adr_type_t`.

A few common traits simplify access to `adr_data_t` objects. The first is that, except for the structure and array derived types (not enumerations), all `adr_data_t` values are immutable. They are assigned a value when they are created, and may not be changed thereafter.

Another is that all `adr_data_t` values are reference counted. Sometimes data structures need to be used by multiple consumers simultaneously, or simply retained for subsequent use. Reference counting is a cheap way to cut down on the cost of copying large data structures and the complexity of handling allocation failures. Though the reference counting is thread-safe, there is no other locking, which is not a problem for an immutable `adr_data_t`. Though the

value of a non-immutable adr_data_t may be modified post-creation, the convention used throughout rad and its associated libraries is that once visibility of a adr_data_t has spread past its creator, it may no longer be modified. This eliminates the need for additional synchronization.

```
adr_data_t *adr_data_ref(adr_data_t *data);
void adr_data_free(adr_data_t *data);
```

The reference count on the adr_data_t data is incremented with adr_data_ref. For convenience, adr_data_ref returns data. Symmetrically, the reference count on the adr_data_t data is decremented with adr_data_free. As the name implies, this may result in data being freed; after calling adr_data_free the caller must not access data in any way. Neither adr_data_ref nor adr_data_free can fail.

A third trait is that interfaces that accept adr_data_t values take ownership of the caller's reference on the adr_data_t. If the caller needs to refer to the adr_data_t after passing a pointer to it to a libadr interface, it must first secure an additional reference with adr_data_ref. Interfaces that return adr_data_t that are referenced by other adr_data_t do not increase the reference count on the returned adr_data_t. The returned value is guaranteed to persist only as long as the caller retains a reference on the referring adr_data_t, or if the caller uses adr_data_ref to acquire its own reference on the returned adr_data_t. The net result is that in the common case where a adr_data_t does not have multiple simultaneous consumers, libadr consumers need not perform any explicit reference counting at all. They can naively allocate and free adr_data_t values as if they were any other data structure. Note also that the adr_data_t implementation can therefore optimize for the case where the reference count is 1.

Lastly, many adr_data_t management routines rely on dynamic memory allocation, which means that proper error handling is essential. To increase the clarity and maintainability of adr_data_t consumers, and reduce the likelihood of mishandling errors, libadr interfaces explicitly accept NULL adr_data_t inputs and fail in sympathy. This means that a libadr consumer can perform a large number of operations on the instances of adr_data_t, checking only the final result for failure. Additionally, if a libadr routine is going to fail for any reason, references to a non-NULL adr_data_t passed to the routine is released. In other words, no special clean-up is needed when a libadr routine fails.

Allocating adr_data_t Values

The first phase in the lifecycle of a adr_data_t is allocation. For each ADR type, there is at least one allocation routine. The arguments to an allocation routine depend on the type. In the case of mandatorily immutable types, allocation implies initialization, and their allocation routines take as arguments the value the adr_data_t is to have. Structures and arrays each have a single generic allocation routine that takes a adr_type_t* specifying the type of the structure or array. A adr_data_t is assigned values using a separate set of routines.

All allocation routines return a non-NULL adr_data_t * on success, or NULL on failure.

Note – The allocation and initialization routines for immutable types may elect to return a reference to a shared `adr_data_t` for a commonly used value, for example, boolean true or false. This substitution should be undetectable by `adr_data_t` consumers who correctly manage `adr_data_t` reference counts and respect the immutability of these types.

Allocating Strings

```
adr_data_t *adr_data_new_string(const char *s, lifetype_t lifetime);
```

Allocates a new string `adr_data_t`, initializing it to the NUL-terminated string pointed to by s. If s is `NULL`, `adr_data_new_string` will fail.

The value of the `lifetime` determines how the string `s` is to be used.

`LT_COPY`	`adr_data_new_string` must allocate and make a copy of the string pointed to by s. This copy will be freed when the `adr_data_t` is freed.
`LT_CONST`	The string pointed to by s is a constant that will never be changed or deallocated. Therefore, `adr_data_new_string` need not copy the string; it can instead refer directly to s indefinitely. This is the recommended lifetime value when passing a string literal to `adr_data_new_string`.
`LT_FREE`	The string pointed to by s was dynamically allocated using `malloc` and is no longer needed by the caller. `adr_data_new_string` will ensure that the string is eventually freed. It may choose to use the string directly instead of making a copy of it. Obviously, this lifetime value should never be used with string literals.

If lifetime is `LT_FREE` and `adr_data_new_string` fails for any reason, s will automatically be freed.

```
adr_data_t *adr_data_new_fstring (const char *format, ...);
```

Allocates a new string `adr_data_t`, initializing it to the string generated by calling `sprintf` on format and any additional arguments provided.

```
adr_data_t *adr_data_new_nstring (const char *s, int count);
```

Allocates a new string `adr_data_t`, initializing it to the first count bytes of `s`.

Allocating boolean

```
adr_data_t *adr_data_new_boolean (boolean_t b);
```

Allocates a new boolean `adr_data_t`, initializing it to the boolean value specified by `b`.

Allocating Numeric Types

```
adr_data_t *adr_data_new_integer (int i);

adr_data_t *adr_data_new_long (long long l);

adr_data_t *adr_data_new_uinteger (unsigned int ui);

adr_data_t *adr_data_new_ulong (unsigned long ul);

adr_data_t *adr_data_new_float (float f);

adr_data_t *adr_data_new_double (double d);
```

Allocates a new integer, (`int`), `long`, `uinteger`, `ulong`, `float`, or `double` `adr_data_t`, respectively, initializing it to the value of the single argument provided.

Allocating Times

```
adr_data_t *adr_data_new_time (long long sec, int nano);

adr_data_t *data_new_time_ts (time_t t);

adr_data_t *adr_data_new_time_now (void );
```

Allocates a new `time` `adr_data_t`, initializing it to the argument, if any, provided.

Allocating Opaques

```
adr_data_t *adr_data_new_opaque (void *buffer, int length, lifetime_t lifetime);
```

Allocates a new opaque `adr_data_t`, initializing it to the length bytes found at buffer. How `adr_data_new_opaque` uses buffer depends on lifetime, which takes on the same meanings as it does when used with `adr_data_new_string`.

Allocating Secrets

```
adr_data_t *data_new_password (const char *p);
```

Allocates a new secret `adr_data_t`, initializing it to the contents of the `NULL`-terminated 8-bit character array pointed to by `p`. The secret type is used to hold sensitive data such as passwords. Client/server implementations may take additional steps to protect the content of password data, for example, zeroing buffers after use.

Allocating Names

```
adr_data_t *adr_data_new_name (adr_name_t *name);
```

Allocates a new name `adr_data_t`, initializing it to the value of name. `adr_name_t` types are reference counted; the reference on name held by the caller is transferred to the resulting `adr_data_t` by the call to `adr_data_new_name`. A caller that needs to continue using name should secure an additional reference to it before calling `adr_data_new_name`. If `adr_data_new_name` fails for any reason, the caller's reference to name will be released.

Allocating Enumerations

```
adr_data_t *adr_data_new_enum (adr_type_t *type, int value);

adr_data_t *adr_data_new_enum_byname (adr_type_t *type, const char * name);
```

The two ways to allocate an enumeration `adr_data_t` both require that the `adr_type_t` of the enumeration be specified. The first form, `adr_data_new_enum`, takes a scalar value as an argument and initializes the enumeration `adr_data_t` to the enumerated value that was assigned (implicitly or explicitly) that scalar value. The second form, `adr_data_new_enum_byname`, takes a pointer to a string as an argument and initializes the enumeration `adr_data_t` to the enumerated value that has that name. If value does not correspond to an assigned scalar value or name does not correspond to an enumerated value name, the respective allocation routine fails.

The nature of an enumeration is that all possible values are known. Enumerated types generated by `radadrgen` have singleton `adr_data_t` values that will be returned by `adr_data_new_enum` and `adr_data_new_enum_byname`. For efficiency and to reduce the error handling that needs to be performed at runtime, these values have defined symbols that may be referenced directly.

The value of type must be an enumeration data-type.

Allocating Unions

```
adr_data_t *adr_data_new_union (adr_type_t *uniontype,
     adr_data_t *discriminator, adr_data_t *value);
```

Allocates a new union `adr_data_t`, initializing it with the discriminator and arm data value provided.

Allocating Structures

```
adr_data_t *adr_data_new_struct (adr_type_t *type);
```

Allocates an uninitialized structure `adr_data_t` of type type. Any post-allocation initialization that occurs must be consistent with type.

The value of type must be a structured type.

Allocating Arrays

```
adr_data_t *adr_data_new_array (adr_type_t *type, int size);
```

Allocates an empty array `adr_data_t` of type `type`. Arrays will automatically adjust their size to fit the amount of data placed in them. size can be used to initialize the size of the array if it is known beforehand.

The value of type must be an array type.

Accessing Simple adr_data_t Values

`rad/adr.h` defines macros that behave like the following prototypes:

```
const char *adr_data_to_string(adr_data_t *data);

int adr_data_to_integer(adr_data_t *data);

unsigned int adr_data_to_uinteger(adr_data_t *data);

long long adr_data_to_longint(adr_data_t *data);

unsigned long long adr_data_to_ulongint(adr_data_t *data);

boolean_t adr_data_to_boolean(adr_data_t *data);

adr_name_t *adr_data_to_name(adr_data_t *data);

const char *adr_data_to_secret(adr_data_t *data);

float adr_data_to_float(adr_data_t *data);

double adr_data_to_double(adr_data_t *data);

const char * adr_data_to_opaque(adr_data_t *data);

long long adr_data_to_time_secs(adr_data_t *data);

int adr_data_to_time_nsecs(adr_data_t *data);
```

In all cases, pointer return values will point to data that is guaranteed to exist only as long as the caller retains their reference to the data parameter.

Additionally, the following functions are provided for interpreting enumeration values:

```
const char *adr_enum_tostring(adr_data_t *data);
int adr_enum_tovalue(adr_data_t *data);
```

`adr_enum_tostring` maps data to the value's string name. `adr_enum_tovalue` maps data to its scalar value.

The behavior is undefined if a macro or function is called on a `adr_data_t` of the wrong type.

The following functions are used to access union data:

```
adr_data_t *adr_union_get(adr_data_t *uniondata);

adr_data_t *adr_union_get_arm(adr_data_t *uniondata);
```

`adr_union_get` returns the data stored in the arm selected by the discriminator. `adr_union_get_arm` returns the value of the discriminator.

Manipulating Derived Type adr_data_t

Structure and array derived types are assigned no value when they are allocated. As a best practice, you should assign some value to them before use; in the case of structured types with non-nullable fields, it is required. In either case, once a reference to a derived type is shared, it may no longer be modified.

Manipulating Array adr_data_t Values

`rad/adr.h` defines array-access macros that behave like the following prototypes:

```
int adr_array_size(adr_data_t *array);
adr_data_t *adr_array_get(adr_data_t *array, int index);
```

`adr_array_size` returns the number of elements in array. `adr_array_get` returns the index element of array. The `adr_data_t` returned by `adr_array_get` is valid as long as the caller retains its reference to array; if it is needed longer, the caller should take a hold on the `adr_data_t` (see "`adr_data_t` Type" on page 32). If the index element of array has not been set, the behavior of `adr_array_get` is undefined.

The following functions modify arrays:

```
int adr_array_add(adr_data_t *array, adr_data_t * value);
```

`adr_array_add` adds value to the end of array. As described in "`adr_data_t` Type" on page 32, the caller's reference to value is transferred to the array. `adr_array_add` might need to allocate memory and can therefore fail. When `adr_array_add` succeeds, it returns 0. When `adr_array_add` fails, it will return 1 and array will be marked invalid. For more information, see "Validating `adr_data_t` Values" on page 39.

```
void adr_array_remove(adr_data_t *array, int index);
```

`adr_array_remove` removes the index element from array. The array's reference count on the element at index is released, possibly resulting in its deallocation. All elements following index in array are shifted to the next lower position in the array, for example, element index+1 is moved to index. The behavior of `adr_array_remove` is undefined if index is greater than or equal to the size of array as returned by `adr_array_size`.

```
int adr_array_vset(adr_data_t *array, int index, adr_data_t * value);
```

`adr_array_vset` sets the index element of array to value. If an element was previously at index, the reference on that element held by the array is released. `adr_array_vset` may need to allocate memory and can therefore fail. When `adr_array_vset` succeeds, it returns 0. When `adr_array_vset` fails, it will return 1 and array will be marked invalid. For more information, see "Validating `adr_data_t` Values" on page 39.

Manipulating the Structure of a adr_data_t Type

The primary interface for accessing a `adr_data_t` structure is `adr_struct_get`:

```
adr_data_t *adr_struct_get(adr_data_t *struct, const char *field);
```

`adr_struct_get` returns the value of the field named field. If the field is nullable and has no value or if the field hasn't been given a value (that is the structure was incompletely initialized), `adr_struct_get` returns `NULL`. The `adr_data_t` returned by `adr_struct_get` is valid as long as the caller retains its reference to `struct`. If it is needed longer the caller should take a hold on the `adr_data_t`. If `struct` does not have a field named field, the behavior of `adr_struct_get` is undefined.

The primary interface for writing to a `adr_data_t` structure is `adr_struct_set`:

```
void adr_struct_set(adr_data_t *struct, const char *field, adr_data_t *value);
```

`adr_struct_set` writes value to the field named field. It If field previously had a value, the reference on that value held by the structure is released. If `struct` does not have a field named field, or if the type of value does not match that of the specified field the behavior of `adr_struct_set` is undefined.

Validating adr_data_t Values

`libadr` provides a rich environment for examining and manipulating typed data. However, unlike C's native typing system, the compiler is unaware of `libadr` type relationships and is therefore unable to perform static type-checking at compile time. All type checking must be performed at runtime.

The most useful of the type-checking tools provided by `libadr` is `adr_data_verify`:

```
boolean_t adr_data_verify(adr_data_t *data, adr_type_t *type, boolean_t recursive);
```

`adr_data_verify` takes a `adr_data_t` to type-check and a `adr_type_t` to type-check against, H can be instructed to check only the `adr_data_t` data or data and the transitive closure of every `adr_data_t` it references. `adr_data_verify` returns `B_TRUE` if data matches type, and `B_FALSE` if not. If type is `NULL`, data is tested against the type it claims to be. Although this method is not a good idea for input validation, it can be useful for error handling.

In order for data to be verified as type type, the following must be true:

- data must not be `NULL`.
- data must claim to be of type `type`.
- If `type` is an enumeration, data must be a value in that enumeration.
- If `data` is an array, it must be not have been marked invalid by a failed `adr_array_add` or `adr_array_vset` operation.

- If `data` is an array, it must have no `NULL` elements.
- If `data` is an array and recursive is true, each element of the array must satisfy these criteria given the array's element type.
- If `data` is a structure, every non-nullable field must have a value, that is, be non-`NULL`.
- If `data` is a structure and recursive is true, every non-`NULL` field value must satisfy these criteria considering the field's type.

Obviously, `adr_data_verify` is useful when validating input from an untrusted source. A second, less obvious application of `adr_data_verify` is as a powerful error-handling tool. Suppose you are writing a function that needs to return a complex data value. A traditional way of implementing it would be to check each call for failure individually, as shown in the following example.

EXAMPLE 4–1 Traditional Error Handling

```
adr_data_t *tmp, *name, *result;
if ((name = adr_data_new_struct(name_type)) == NULL) {
/* handle failure */
}
if ((tmp = adr_data_new_string("Jack")) == NULL) {
/* handle failure */
}
adr_struct_set(name, "first", tmp);
if ((tmp = adr_data_new_string("O'Neill")) == NULL) {
/* handle failure */
}
adr_struct_set(name, "last", tmp);
if ((record = adr_data_new_struct(record_type)) == NULL) {
/* handle failure */
}
adr_struct_set(record, "name", name);
/* ...and so on */
```

This approach is difficult to implement and difficult to maintain. It is more likely to have a flaw in it than the allocations it is testing are to fail. Instead, using `adr_data_verify` and the error handling behaviors described in "`adr_data_t` Type" on page 32, the entire non-truncated function can be reduced to the method shown in the following example.

EXAMPLE 4–2 Error Handling With adr_data_verify

```
adr_data_t *name = adr_data_new_struct(name_type);
adr_struct_set(name, "first", adr_data_new_string("Jack"));
adr_struct_set(name, "last", adr_data_new_string("O'Neill"));
adr_data_t *record = adr_data_new_struct(record_type);
adr_struct_set(record, "name", name);
adr_struct_set(record, "rank", adr_data_new_enum_byname("COLONEL"));
adr_struct_set(record, "l_count", adr_data_new_integer(2));

if (!adr_data_verify(record, NULL, B_TRUE)) { /* Recursive type check */
            adr_data_free(record);
            return (NULL); /* NULL means something failed */
}
```

EXAMPLE 4–2 Error Handling With adr_data_verify *(Continued)*

```
return (record); /* Non-NULL means success */
```

An important limitation to this technique is the possibility for structure fields to be nullable, and the `NULL` indicating that the field has no value is indistinguishable from the `NULL` that indicates that the allocation of that field's value failed. In such cases, explicitly testing each nullable value's allocation is necessary. Even with such explicit checks, however, the net savings in complexity can be substantial.

ADR Object Name Operations

`libadr` supports ADR object names by providing a `adr_name_t` type and a suite of routines for creating and inspecting them. Consumers needing to operate on object names should include the `rad/adr_name.h` header file:

```
#include <rad/adr_name.h>
```

This file contains definitions for all the ADR-name related functionality provided by `libadr`.

adr_name_t Type

The `adr_name_t` type represents an object name. The internal structure of an `adr_name_t` is private. All operations on a `adr_name_t` are performed using accessor functions provided by `libadr`. Like `adr_data_t` values, `adr_name_t` values are immutable and reference counted. The following functions are provided for handling `adr_name_t` reference counts:

```
adr_name_t *adr_name_hold(adr_data_t *name);
void adr_name_rele(adr_name_t *name);
```

The reference count on the `adr_name_t` name is incremented with `adr_name_hold`. For convenience, `adr_name_hold` returns name. Symmetrically, the reference count on the `adr_name_t` name is decremented with `adr_name_rele`. When then last reference on an `adr_name_t` is released, the name is freed; after calling `adr_name_rele` the caller must not access name in any way. Neither `adr_name_hold` nor `adr_name_rele` can fail.

Creating adr_name_t Type

ADR names are composed of a domain and a set of key/value pairs. Two functions are provided that take exactly those arguments and return an `adr_name_t`:

```
adr_name_t *adr_name_create(const char *domain, int count,
     const char * const *keys, const char * const *values);

adr_name_t *adr_name_vcreate(const char *domain, int count, ...);
```

Both forms take a domain argument, which should be a reverse-dotted domain name, and the number of key/value pairs as count. The two differ in how the key/value values are communicated. In the first form, `adr_name_create`, two `char *` arrays are provided, one for keys and the other for values, as shown in the following example.

EXAMPLE 4–3 adr_name_create

```
const char *keys[] = { "key1", "key2" };
const char *values[] = { "value1", "value2" };
name = adr_name_create("com.example", 2, keys, values);
```

In the second form, `adr_name_vcreate`, keys and values are provided as alternating varargs. The previous example written using `adr_name_vcreate` would look like the following example.

EXAMPLE 4–4 adr_name_vcreate

```
name = adr_name_vcreate("com.example", 2, "key1", "value1", "key2", "value2");
```

If either routine fails to create the `adr_name_t`, it will return `NULL`. All data provided to `adr_name_create` is copied and can subsequently be modified or freed without affecting existing `adr_name_t` types.

Sometimes, it is convenient to start with an existing ADR name and append additional key/value pairs to form the desired name. For this situation, `libadr` provides "compose" analogues to the previously described creation routines:

```
adr_name_t *adr_name_compose(const adr_name_t *name, int count,
     const char * const *keys, const char * const *values);

adr_name_t *adr_name_vcompose(const adr_name_t *name, int count, ...);
```

These two functions behave the same as their corresponding `adr_create_*` routine except that they take an `adr_name_t` instead of a domain. The resulting `adr_name_t` has the domain from name, and merges the key/value pairs found on name with those provided as arguments. The key/value arguments to the compose operations must not specify keys already present on name.

The `adr_name_t` name passed to the compose operations is not referenced by their return values. In many cases, it will be used as an argument to these functions again and again. Unlike `adr_data_t` allocation routines, `adr_name_compose` and `adr_name_vcompose` do not consume the caller's reference to the provided name.

Inspecting adr_name_t Type

`adr_name_t` types are immutable, so all operations on them are read-only. The two most common operations one needs to perform on an `adr_name_t` are obtaining the name's domain and obtaining the value associated with a particular key.

```
const char *adr_name_domain(const adr_name_t *name);
const char *adr_name_key(const adr_name_t *name, const char *key);
```

adr_name_domain returns name's reverse-dotted domain as a string. The string returned is part of name and therefore must not be modified or freed, and must not be accessed after the caller's reference on name has been released. Likewise, adr_name_key returns the value associated with key. The string returned by adr_name_key is subject to the same restrictions as the return value of adr_name_domain.

The two functions for comparing adr_name_t types are:

```
int adr_name_cmp(const adr_name_t *name1, const adr_name_t *name2);

boolean_t adr_name_match(const adr_name_t *name, const adr_name_t *pattern);
```

adr_name_cmp compares two adr_name_t types, returning 0 if the name1 and name2 are equal (that is, if the two names have the same names and the same keys, and each key has the same value on both names). It returns an integer less than 0 if name1 is less than name2, or and integer greater than 0 if name1 is greater than name2.

adr_name_match is a pattern-matching operation. The adr_name_t pattern is treated as a collection of attributes against which name is compared. adr_name_match returns B_TRUE if and only if the domains of name and pattern are equal, and every key present in pattern is present in name and has the same value. While normally an adr_name_t must have a domain and at least one key/value pair, pattern is permitted to have no key/value pairs and an empty domain. An empty pattern domain is considered a wildcard that matches any name domain.

String Representation

It is sometimes necessary to represent, either in human-readable output or in persistent storage, an ADR object name as a string. libadr provides routines for converting to and from a canonical string form.

```
adr_name_t *adr_name_fromstr(const char *str);

char *adr_name_tostr(const adr_name_t *name);
```

adr_name_fromstr takes a string and returns the corresponding adr_name_t. It behaves like an allocation routine, as described in "Creating adr_name_t Type" on page 41. If the string isn't a valid name, adr_name_fromstr returns NULL.

adr_name_tostr takes an adr_name_t and formats it in string form. The return value is allocated using malloc and should be freed when the caller is done with it. adr_name_tostr will return NULL if it is unable to allocate memory for its return value.

API Management

libadr provides support for defining APIs in rad/adr_object.h. Defining an API is a complex task. The only supported way to define an API is to do so in the ADR IDL and to generate the definition using radadrgen.

The important type defined in rad/adr_object.h is type adr_object_t. While the constituent pieces of an API definition should be considered implementation details, the end product, the API itself, is of prime interest to the developer. You will never need to create or define an adr_object_t, but when you encounter routines that operate on them, understanding what the type represents is important.

radadrgen-Generated Definitions

Whether you are using libadr in a C-based client or as part of writing a rad server module, you will need to understand the data definitions generated by radadrgen. Fortunately, the definitions are the same in both environments.

Running radadrgen

radadrgen is instructed to produce definitions for C/libadrconsumers by using its -c option, as shown in the following example.

EXAMPLE 4–5 Invoking radadrgen

```
$ radadrgen -c output_dir example.xml
```

The -c option produces two files, api_APINAME.h and api_APINAME_impl.c in the *output_dir*, where APINAME is the value of the name attribute of the API document's api element. api_APINAME_impl.c contains the implementation of the interfaces and data types defined by the API. It should be compiled and linked with the software needing those definitions.

api_APINAME.h externs the specific symbols defined by api_APINAME_impl.c that consumers will need to reference, and should be #included by those consumers. api_APINAME.h contains no data definitions itself and may be included in as many places as necessary. The definitions api_APINAME_impl.c are 100% data and are statically initialized. There are no initialization functions to be called. Neither file should be modified.

For each derived type TYPE, whether enumeration or structure, defined in the API, a adr_type_t named t__TYPE (two underscores) representing that type is generated and externed by the header file. If an array of that type is used anywhere in the API, a adr_type_t named t_array__TYPE (one underscore, two underscores) representing that array type is generated and externed. For each interface INTERFACE defined in the file, an adr_object_t named interface_INTERFACE is defined and externed.

EXAMPLE 4–5 Invoking radadrgen *(Continued)*

For each value `VALUE` of an enumeration named `TYPE` , a `adr_data_t` named `e__TYPE_VALUE` is defined and externed. These `adr_data_t` values are marked as constants and are not affected by `adr_data_ref` or `adr_data_free`.

Example radadrgen output

When `radadrgen` is run on the Example 3–10 given in the ADR chapter two files result. One, `api_example_impl.c`, holds the implementation of the `GrabBag` interface and data types it depends on, and should be simply be compiled and linked with the `GrabBag` consumer. The other, `api_example.h`, exposes only the relevant symbols defined by `api_example_impl.c` and should be included by consumers of the GrabBag interface and its related types as shown in the following example.

EXAMPLE 4–6 Sample radadrgen-Generated C Header File

```
#include <rad/adr.h>
#include <rad/adr_object.h>

extern adr_type_t t__Mood;
extern adr_data_t e__Mood_IRREVERENT;
extern adr_data_t e__Mood_MAUDLIN;
extern adr_type_t t__SqrtError;
extern adr_type_t t__StringInfo;
extern adr_type_t t__MoodStatus;
extern adr_object_t interface_GrabBag;
```

The function of `api_GrabBag` is discussed later in this document, but the purpose of the other definitions in this file should be clear. A consumer needing to create a MoodStatus structure indicating the mood is IRREVERENT and has changed would issue the instructions shown in the following example.

EXAMPLE 4–7 Consuming radadrgen-Generated Definitions

```
status = adr_data_new_struct(&t__MoodStatus);
adr_struct_set(status, "mood", e__Mood_IRREVERENT);
/* adr_struct_set(status, "mood", adr_data_new_enum_byname(&t__Mood, "IRREVERENT")); */
adr_struct_set(status, "changed", adr_data_new_boolean(B_TRUE));

if (!adr_data_verify(status, NULL, B_TRUE)) {
        ...
```

In addition to showing how to use the type definitions, this example also illustrates the multiple ways of referencing an enumerated value. Using the defined symbols is faster and can be checked by the compiler. The commented-out line uses `adr_data_new_enum_byname` which offers flexibility that could be useful in some situations but necessarily defers error checking

until runtime. For example, if the programmer mistyped the value “IRREVERENT”, it would not be detected until the code was run. Obviously, using the enumerated value symbols when possible is preferable.

◆ ◆ ◆ CHAPTER 5

Client Libraries

rad provides support for three client language environments: Java/JMX, Python, and C.

Java/JMX Client

Connecting to the rad Server

Connections to a rad server in Java are made through JMX. A JMX connection is established with the following:

- A javax.management.remote.JMXService URL that identifies the protocol to be used, and a host, port, and path where appropriate
- A Map<String, Object> of protocol-specific options where appropriate

Several protocols for connecting to a rad server are supported and are defined in the com.oracle.solaris.rad.jmx class. Unless explicitly mentioned, a host, port, path, or options is not applicable.

These protocols are:

RadConnector.PROTOCOL_UNIX

A local UNIX domain socket connection. The path is the fully qualified name of the socket.

EXAMPLE 5–1 RadConnector.PROTOCOL_UNIX

```
JMXServiceURL url = new JMXServiceURL(RadConnector.PROTOCOL_UNIX,
  "", 0, "/system/volatile/rad/radsocket");
JMXConnector connector = JMXConnectorFactory.connect(url);
```

RadConnector.PROTOCOL_TCP

A local/remote TCP connection. A host and/or port may be specified.

EXAMPLE 5–2 RadConnector.PROTOCOL_TCP

```
JMXServiceURL url = new JMXServiceURL(RadConnector.PROTOCOL_TCP,
    "myhost", 0);
JMXConnector connector = JMXConnectorFactory.connect(url);
```

RadConnector.PROTOCOL_TLS

A local/remote TLS connection. A host and/or port may be specified.

Expected parameters:

`RadConnector.KEY_TLS_TRUSTSTORE` (required)
: The full path to a local `rad` truststore file

`RadConnector.KEY_TLS_TRUSTPASS` (required)
: The password for the local `rad` truststore

`RadConnector.KEY_TLS_RADMANAGER` (required)
: An instance of `com.oracle.solaris.rad.RadTrustManager` for key management

EXAMPLE 5–3 RadConnector.PROTOCOL_TLS

```
Map<String, Object> env = new HashMap<String, Object>();
env.put(RadConnector.KEY_TLS_TRUSTSTORE, "/etc/myapp/truststore");
env.put(RadConnector.KEY_TLS_TRUSTPASS, "trustpass");

JMXServiceURL url = new JMXServiceURL(
    RadConnector.PROTOCOL_TLS, host, 0);

JMXConnector connector = JMXConnectorFactory.newJMXConnector(url, null);

for (;;) {
    RadTrustManager mtm = new RadTrustManager();
    env.put(RadConnector.KEY_TLS_RADMANAGER, mtm);

    try {
        connector.connect(env);
        break;
    } catch (IOException e) {
        X509Certificate[] chain = mtm.getBadChain();
        if (chain == null) {
            throw e;
        }
    }
}
```

RadConnector.PROTOCOL_PRIVATE

A local connection to a `rad` instance private to this process, spawned when the connection is established. The resulting `rad` communicates through its stdin/stdout.

Expected parameters:

`RadConnector.PRIVATE_ROOT` (String, optional)
A full path to prefix the to each relatively named module in the RadConnector.PRIVATE_MODULES parameter, if specified

`RadConnector.PRIVATE_MODULES` (String[], optional)
A list of modules to load, as with `/usr/lib/rad/rad -M` module

`RadConnector.PRIVATE_AUXARGS` (String[], optional)
Additional arguments to pass to the spawned `rad` instance

EXAMPLE 5–4 RadConnector.PROTOCOL_PRIVATE

```
Map<String, Object> env = new HashMap<String, Object>();
String[] auxargs = { "-d", "-e", "30" };
String[] modules = { "mod_usermgmt.so", "mod_nameservice.so" };

env.put(RadConnector.PRIVATE_AUXARGS, auxargs);
env.put(RadConnector.PRIVATE_MODULES, modules);
env.put(RadConnector.PRIVATE_ROOT, "/usr/share/modules");

JMXServiceURL url = new JMXServiceURL(
    RadConnector.PROTOCOL_PRIVATE, "", 0);

JMXConnector connector = JMXConnectorFactory.newJMXConnector(url, env);
```

RadConnector.PROTOCOL_ZONESBRIDGE

A connection to a non-global zone's local UNIX `rad` instance, through an existing local or remote `rad` connection to its global zone. The name of the non-global zone is specified as the host. The non-global zone user is specified as the path.

Expected parameters:

`RadConnector.KEY_ZONESBRIDGE_MXBEAN`
(`com.oracle.solaris.rad.zonesbridge.IOMXBean`, required)

An `IOMXBean` from an existing `rad` connection to the global zone, retrieved with a domain "`com.oracle.solaris.rad.zonesbridge`" and a "`type=IO`" key/value pair.

EXAMPLE 5–5 RadConnector.PROTOCOL_ZONESBRIDGE

```
// Create a connection to some host
MBeanServerConnection mbsc = ...

ObjectName zioName = new ObjectName(
  "com.oracle.solaris.rad.zonesbridge", "type", "IO");
IOMXBean zio = JMX.newMXBeanProxy(mbsc, zioName, IOMXBean.class);

// The zone to connect to
String zone = "nerd-vpn";

// The non-global-zone user, or "" to connect as root
String zoneUser = "talley";
```

EXAMPLE 5–5 RadConnector.PROTOCOL_ZONESBRIDGE *(Continued)*

```
// Create a connection to the non-global zone on the connected host
JMXServiceURL zUrl = new JMXServiceURL(
    RadConnector.PROTOCOL_ZONESBRIDGE, zone, 0, "/" + zoneUser);

Map<String, Object> env = new HashMap<String, Object>();
env.put(RadConnector.KEY_ZONESBRIDGE_MXBEAN, zio);

JMXConnector connector = JMXConnectorFactory.connect(zUrl, env);
```

radadrgen Usage

In the Java/JMX environment, elements declared in the module specification are translated to Java classes and interfaces. The tool that does this translation is `/usr/bin/radadrgen`:

```
/usr/bin/radadrgen [ -N ] -j dir [ -i ] spec.xml
```

The Java classes and interfaces generated by `radadrgen` are as follows, where package is determined from the `<api>` name or a supplied pragma:

Element	radadrgen-Generated Class	Description
`<interface>`	*package.interface*`MXBean`	MXBean interface, implemented by object returned by the `rad` server
`<struct>`	*package.struct*	`struct` interface
	package.struct`Impl`	`struct` implementation (if `-i` is used)
`<union>`	*package.union*	`union` interface
	package.union`Impl`	`union` implementation
`<enum>`	*package.enum*	`enum` class

See the section on object naming for details.

Enums

For example, suppose you have the `rad` module specification `example.xml`:

```
<api xmlns="http://xmlns.oracle.com/radadr"
  name="com.example.foo">

  <enum name="Color">
    <value name="Red" />
    <value name="Green" />
```

```
    <value name="Blue" />
  </enum>
  ...
</api>
```

Calling `/usr/bin/radadrgen -j dir example.xml` generates a native Java enum class:

```
package com.example.foo;

public enum Color {
    Red,
    Green,
    Blue,
}
```

Structured Types

Adding a <struct> to the module API produces a Java interface that models the structured type. For example:

```
<struct name="Person">
  <field type="string" name="name" />
  <field type="integer" name="age" />
  <field typeref="Color" name="favoriteColor" />
</struct>
```

resulting Java interface:

```
package com.example.foo;


public interface Person {
    String getName();
    int getAge();
    Color getFavoriteColor();
}
```

If the -i is also passed to `radadrgen`, an implementation of the `Person` interface is also generated:

```
package com.example.foo;

public class PersonImpl implements Person {
    private String name_;
    private int age_;
    private Color favoriteColor_;

    public PersonImpl() {
    }

    public PersonImpl(String name, int age, Color favoriteColor) {
        name_ = name;
        age_ = age;
        favoriteColor_ = favoriteColor;
    }
```

```
    public String getName() {
        return name_;
    }

    public int getAge() {
        return age_;
    }

    public Color getFavoriteColor() {
        return favoriteColor_;
    }

    public void setName(String arg) {
        name_ = arg;
    }

    public void setAge(int arg) {
        age_ = arg;
    }

    public void setFavoriteColor(Color arg) {
        favoriteColor_ = arg;
    }
}
```

This process can be useful in client code to quickly create and use a basic object that implements a structured type's interface.

Unions

When adding a discriminated union to a module API, `radadrgen` produces a Java interface that models the union. For example:

Discriminated Union:

```
<union name="ColorData" typeref="Color">
  <arm value="Red" type="string" />
  <arm value="Green" type="integer" />
  <arm value="Blue" type="float" />
</union>
```

Resulting Java interface:

```
package com.example.foo;


public interface ColorData {
    Color getArm();

    String getData_Red();
    Integer getData_Green();
    Float getData_Blue();
    ...
}
```

To use a `union`, the `getArm`() method is first called to determine which of the union arms is active. Based on the return value of that method, the appropriate getData_*() method can be called to get the data encapsulated in the union.

To create a union, the generated Java interface also includes several static convenience classes, as shown in the following example.

```
public interface ColorData {
    ...

    static class arm_Red extends ColorDataImpl {
        private String armdata_;
        public arm_Red(String armdata) {
            super(Color.Red);
            armdata_ = armdata;
        }
        @Override
        public String getData_Red() { return armdata_; }
    }

    static class arm_Green extends ColorDataImpl {
        ...
    }

    static class arm_Blue extends ColorDataImpl {
        ...
    }
}
```

These inner classes provide a quick way to quickly create and use a basic object that implements the generated union interface for a given union arm.

Interfaces

Interfaces are the reason that modules exist. For an `<interface>` added to a module, `radadrgen` produces a Java `MXBean` interface. For example:

Interface:

```
<interface name="Population">
  <property name="groupName" access="rw" type="string"/>
  <property name="people" access="ro">
    <list typeref="Person" />
  </property>
  <method name="add">
    <argument typeref="Person" name="person" />
  </method>
</interface>
```

Resulting Java `MXBean` interface:

```
package com.example.foo;
```

```
public interface PopulationMXBean {
    String getgroupName();

    void setgroupName(String groupName);

    java.util.List<Person> getpeople();

    void add(Person person);
}
```

This `MXBean` is implemented by the objects returned by the `rad` server:

```
// Retrieve the Population object
ObjectName oName = new ObjectName("com.example.foo",
    "type", "Population");
PopulationMXBean pop = JMX.newMXBeanProxy(mbsc,
    oName, PopulationMXBean.class);

// Access a property
List<Person> people = pop.getpeople();

// Call a method
pop.add(new PersonImpl("talley", Color.GREEN));
```

Caveats

Note the following cautions:

Caution – To eliminate the ambiguity inherent to the JMX `ObjectName` quoting rules, all key values returned by interfaces that return `ObjectNames` are quoted by the JMX client connector before being passed to the caller. For compatibility, the JMX client connector will accept object names with unquoted key values.

Python Client

Modules

The `rad` Python implementation provides the modules described in this section to facilitate client implementation.

client

Provides classes and methods which fully implement the `rad` protocol in Python.

util

Provides an Authentication class and utility methods for connecting a Python client to `rad`.

`rad.util RadAuth`

A class which fetches and caches a handle to an authentication object along with some convenience methods for manipulating it.

The following methods can be used to connect to a `rad` instance using variety of transports.

```
RadConnection connect_unix (string path ,
     string locale);

RadConnection connect_private (string list modules ,
     boolean debug ,
     string map env ,
     string root ,
     string list auxargs ,
     string locale);

RadConnection connect_tls (string host ,
     integer port ,
     string locale);


RadConnection connect_zone(RadConnection rc ,
     string zone ,
     string user ,
     string locale);
```

The resulting `RadConnection` can be used to:

- Locate objects
- Invoke object methods
- Read or write object properties
- Subscribe or unsubscribe to object events

EXAMPLE 5–6 Method Invocation

```
import rad.util

# Connect to a local rad instance.
with rad.util.connect_unix() as rc:

    # Obtain a remote reference to the desired target.
    obj= rc.get_object_s("com.example", [("type", "GrabBag")])

    # Invoke a method on the target.
    res = obj.parseString("a test string")

    # Print the result.
    print "length: " + str(res.length)
```

EXAMPLE 5–7 Attribute Access

```
import rad.util

# Connect to a local rad instance.
with rad.util.connect_unix() as rc:

    # Obtain a remote reference to the desired target.
    obj= rc.get_object_s("com.example", [("type", "GrabBag")])

    # Print the object attribute.
    print "Mood: " + str(obj.mood)
```

EXAMPLE 5–8 Event Subscription

```
import rad.util

with rad.util.connect_unix() as rc:

    # Obtain a remote reference to the desired target.
    obj= rc.get_object_s("com.example", [("type", "GrabBag")])

    # Subscribe to the "moodswings" event
    rc.subscribe(obj, "moodswings")

    while True:
        # Perform a (blocking) read of an event
        ev_obj = obj.read_event()

        print "Received Event:"
        print "mood: " +str(ev_obj.mood)
        print "changed: " +str(ev_obj.changed)
```

◆ ◆ ◆ CHAPTER 6

Module Development

rad is modular in a variety of ways. Modules may deliver new protocols, new transports, or new API definitions and implementations. This section focuses on new API definitions and implementations.

API Definitions and Implementation

Although an API can be constructed manually, using radadrgen to generate the necessary type definitions is much simpler . If requested, radadrgen can also generate stubs for the entry points referenced by the generated types.

Entry Points and Generated Stubs

All entry points take a pointer to the object instance and a pointer to the internal structure for the method or attribute. The object instance pointer is essential for distinguishing different objects that implement the same interface. The internal structure pointer is theoretically useful for sharing the same implementation across multiple methods or attributes, but isn't used and may be removed.

Additionally, all entry reports return a conerr_t. If the access is successful, they should return ce_ok. If the access fails due to a system error, they should return ce_system. If the access fails due to an expected error which should be noted in the API definition, they should return ce_object. If an expected error occurs and an error payload is defined, it may be set in **error*. The caller will unref the error object when it is done with it.

- A method entry point has the type meth_invoke_f:

```
typedef conerr_t (meth_invoke_f)(struct rad_instance *i, struct adr_method *m,
    adr_data_t **result, adr_data_t **args, int count, adr_data_t **error);
```

args is an array of count arguments.

Upon successful return, **result* should contain the return value of the method, if any.

The entry point for a method named `METHOD` in interface `INTERFACE` is named `interface_INTERFACE_invoke_METHOD`.

- An attribute read entry point has the type `attr_read_f`:

```
typedef conerr_t (attr_read_f)(struct rad_instance *i, struct adr_attribute *a,
    adr_data_t **value, adr_data_t **error);
```

Upon successful return, **value* should contain the value of the attribute, if any.

The read entry point for an attribute named `ATTR` in interface `INTERFACE` is named `interface_INTERFACE_read_ATTR`.

- An attribute write entry point has the type `attr_write_f`:

```
typedef conerr_t (attr_write_f)(struct rad_instance *i, struct adr_attribute *a,
    adr_data_t *newvalue, adr_data_t **error);
```

newvalue points to the new value. If the attribute is nullable, *newvalue* can be `NULL`.

The write entry point for an attribute named `ATTR` in interface `INTERFACE` is named `interface_INTERFACE_write_ATTR`.

`rad` explicitly checks the types of all arguments passed to methods and all values written to attributes. Stub implementations can assume that all data provided is of the correct type. Stub implementations are responsible for returning valid data. Returning invalid data results in an undefined behavior.

Global Variables

`boolean_t rad_isproxy`	A flag to determine if code is executing in the main or proxy `rad` daemon.
`rad_container_t *rad_container`	The rad container that contains the object instance.

Module Registration

```
int _rad_init(void *handle);

int rad_module_register(void *handle, int version, rad_modinfo_t *modinfo);
```

A module must provide a `_rad_init`. This is called by the `rad` daemon when the module is loaded and is a convenient point for module initialization including registration. Return `0` to indicate that the module successfully initialized.

`rad_module_register` provides a handle, which is the handle provided to the module in the call to `_rad_init`. This handle is used by the `rad` daemon to maintain the private list of loaded modules. The version indicates which version of the `rad` module interface the module is using. `modinfo` contains information used to identify the module.

Instance Management

```
rad_instance_t *instance_create(adr_name_t *name, rad_object_type *type,
     void *data, void (*)(void *)freef);
```

`instance_create` uses the supplied parameters to create a new instance, with name, of an object of type. data is the user data to store with the instance the freef function is a callback which will be called with the user data when the instance is removed. If the function fails, it returns `NULL`. Otherwise, a valid instance reference is returned. Note that you do not have to call `instance_hold` on a newly created instance, because the reference count is initialized to `1`.

```
rad_instance_t *instance_hold(rad_instance_t *instance);
```

`instance_hold` increments the reference count on instance.

```
void instance_rele(rad_instance_t *instance);
```

`instance_rele` decrements the reference count on instance. If the count reaches 0, then the instance is destroyed.

```
adr_data_t *instance_getname(rad_instance_t *instance);
```

`instance_getname` returns a `adr_data_t *` containing the name of the instance.

```
void * instance_getdata(rad_instance_t *instance);
```

`instance_getdata` returns the user data (supplied in `instance_create`) of the instance.

```
void instance_notify (rad_instance_t *instance, const char *event, long sequence,
     adr_data_t *data);
```

`instance_notify` generates an event on the supplied `instance`. The `seq` is supplied in the event as the sequence number and the payload of the event is provided in data.

Container Interactions

```
conerr_t cont_insert(rad_container_t *container, rad_instance_t *instance,
     long long id);

conerr_t cont_insert_singleton(rad_container_t *container,     adr_name_t *name,
     rad_object_t *object);
```

Create a instance, `rad_instance_t`, using the supplied name and object and then insert into container. If the operation succeeds, `ce_ok` is returned.

```
conerr_t cont_insert_singleton_id(rad_container_t *container, adr_name_t *name,
     rad_object_t *object, long long id);
```

As `cont_insert_singleton_id` but with the ability to specify an id for the created instance. Returns `ce_ok` on success.

```
void cont_remove(rad_container_t *container, rad_instance_t *instance);
```

Remove the `instance` from the container.

```
conerr_t cont_register_dynamic(rad_container_t *container, adr_name_t *pattern,
     rad_dyn_list_t listf, rad_dyn_lookup_t lookupf, void *arg);
```

Register a dynamic container instance manager. The container defines the container in which the instances will be managed. The `pattern` defines the name filter for which this instance manager is responsible.

A typical pattern would define the type of the instance which are managed. For example, `zpat = adr_name_vcreate (DOMAIN, 1, "type", "Zone")` would be responsible for managing all instances with a type of "Zone". `listf` is a user-supplied function which is invoked when objects with the matching pattern are listed. `lookupf` is a user-supplied function which is invoked when objects with the matching pattern are looked up. arg is stored and provided in the callback to the user functions.

Logging

Function	Description
`void rad_log(rad_logtype_t type, const char * format, ...);`	Log a message with type and format to the `rad` log. If the type is a lower level than the `rad` logging level, then the message is discarded.
`void rad_log_alloc()`	Log a memory allocation failure with log level RL_FATAL.
`rad_logtype_t rad_get_loglevel()`	Return the logging level.

Using Threads

Function	Description
`void *rad_thread_arg(rad_thread_t *tp);`	Return the `arg` referenced by the `tp`.

Function	Description
`void rad_thread_ack(rad_thread_t *tp, rad_moderr_t error);`	This function is intended to be used from a user function previously supplied as an argument to `rad_thread_create`. It should not be used in any other context. Acknowledge the thread referenced by `tp`. This process enables the controlling thread, from which a new thread was created using `rad_thread_create`, to make progress. The error is used to update the return value from `rad_thread_create` and should return `rm_ok` for success.
`rad_moderr_t rad_thread_create(rad_threadfp_t fp, void *arg);`	Create a thread to run `fp`. This function will not return until the user function (`fp`) calls `rad_thread_ack. arg` is stored and passed into `fp` as a member of the `rad_thread_t` data. It can be accessed using `rad_thread_arg`.
`rad_moderr_t rad_thread_create_async( rad_thread_asyncfp_t fp, void *arg);`	Create a thread to run `fp. arg` is stored and passed into `fp`.

Synchronization

Function	Description
`void rad_mutex_init(pthread_mutex_t *mutex);`	Initialize a mutex. abort() on failure.
`void rad_mutex_enter(pthread_mutex_t *mutex);`	Lock a mutex. abort() on failure.
`void rad_mutex_exit(pthread_mutex_t *mutex);`	Unlock a mutex. abort() on failure.
`void rad_cond_init(pthread_cond_t *cond);`	Initialize a condition variable, cond. abort(), on failure.

Subprocesses

Function	Description
`exec_params_t *rad_exec_params_alloc`	Allocate a control structure for executing a subprocess.

Function	Description
`void rad_exec_params_free(exec_params_t *params);`	Free a subprocess control structure, params.
`void rad_exec_params_set_cwd(exec_params_t *params, const char *cwd);`	Set the current working directory, `cwd`, in a subprocess control structure, `params`.
`void rad_exec_params_set_env(exec_params_t *params, const char **envp);`	Set the environment, `envp`, in a subprocess control structure, `params`.
`void rad_exec_params_set_loglevel( exec_params_t *params, rad_logtype_t loglevel);`	Set the rad log level, `loglevel`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stdin(exec_params_t *params, int fd);`	Set the stdin file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stdout(exec_params_t *params, int fd);`	Set the stdout file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stderr(exec_params_t *params, int fd);`	Set the stderr file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_forkexec(exec_params_t *params, const char **argv, exec_result_t *result);`	Use the supplied subprocess control structure, `params`, to fork and execute (`execv`) the supplied args, `argv`. If result is not `NULL`, it is updated with the subprocess pid and file descriptor details.
`int rad_forkexec_wait(exec_params_t *params, const char **argv, int *status);`	Use the supplied subprocess control structure, `params`, to fork and execute (`execv`) the supplied args, `argv`. If status is not `NULL`, it is updated with the exit status of the subprocess. This function will wait for the subprocess to terminate before returning.
`int rad_wait(exec_params_t *params, exec_result_t *result, int *status);`	Use the supplied subprocess control structure, params, to wait for a previous invocation of `rad_forkexe` to complete. If result is not `NULL`, it is updated with the subprocess pid and file descriptor details. If status is not `NULL`, it is updated with the exit status of the subprocess. This function will wait for the subprocess to terminate before returning.

Utilities

```
void *rad_zalloc(size_t size);


Return a pointer to a zero-allocated block of size bytes.
char *rad_strndup(char *string,
     size_t length);

int rad_strccmp(const char * zstring,
     const char * cstring,
     size_t length);

int rad_openf(const char *format,
     int oflag,
     mode_tmode,
     ...);

FILE *rad_fopenf(const char *format,
     mode_tmode,
     ...);
```

Locales

Function	Description
`int rad_locale_parse(const char *locale, rad_locale_t **rad_locale);`	Update `rad_locale` with locale details based on locale. If locale is `NULL`, then attempt to retreive a locale based on the locale of the `rad` connection. Returns 0 on success.
`void rad_locale_free(rad_locale_t *rad_locale);`	Free a locale, `rad_locale`, previously obtained with `rad_locale_parse`.

Transactional Processing

There is no direct support for transactional processing within a module. If a transactional model is desirable, then it is the responsibility of the module creator to provide the required building blocks, start_transaction, commit, rollback, and other related processes.

Asynchronous Methods and Progress Reporting

Asynchronous methods and progress reporting is achieved using threads and events. The pattern is to return a token from a synchronous method invocation which spawns a thread to do work asynchronously. This worker thread is then responsible for providing notifications to interested parties events.

Example:

An interface has a method which returns a Task object. The method is called `installpkg` and takes one argument, the name of the package to install.

```
Task installpkg(string pkgname);
```

The Task instance returned by the method, contains enough information to identify a task. Prior to invoking `installpkg`, the client subscribes to a task-update event. The worker thread is responsible for issuing events about the progress of the work. These events contain information about the progress of the task.

In a minimal implementation, the worker thread would issue one event to notify the client that the task was complete and what the outcome of the task was. A more complex implementation would provide multiple events documenting progress and possibly also provide an additional method that a client could invoke to interrogate the server for a progress report.

rad Namespaces

Objects in the `rad` namespace can be managed either as a set of statically installed objects or as a dynamic set of objects that are listed or created on demand.

Static Objects

`rad_modapi.h` declares two interfaces for statically adding objects to a namespace.

`cont_insert`() adds an object to the namespace. In turn, objects are created by calling `instance_create`() with a name, a pointer to the interface the object implements, and a pointer to object-specific callback data. For example:

```
i = instance_create("com.oracle.solaris.user:type=User,name=Kyle",
    &interface_User_svr, kyle_data);
cont_insert(&rad_container, i, error_return);
```

`cont_insert_singleton`() is a convenience routine that creates an object instance for the specified interface with the specified name and adds it to the namespace. The callback data is set to `NULL`.

```
cont_insert_singleton(&rad_container,
    "com.oracle.solaris.user:type=UserManager", &interface_UserManager_svr,
    error_return);
```

rad Module Linkage

Each module is required to provide a function, _rad_init(), for linkage and identification purposes. This function is called before any other function in the module. It is used to initialize the module and register itself with rad.

Note – Within _rad_init(), modules can test the rad_isproxy variable to determine whether this routine is running in the main rad (proxy) daemon.

When rad_isproxy is B_TRUE, modules that depend on running as the authenticated user in the rad slave should return immediately from _rad_init() without further initialization. Modules that do not perform any user-specific or restricted operations should proceed with initialization.

When rad_isproxy is B_FALSE, the module is being initialized in the slave.

EXAMPLE 6–1 Module Initialization and Registration

```
#include <rad/rad_modapi.h>

static rad_modinfo_t modinfo = {"usermgr", "User Management Module"};

int
_rad_init(void *handle)
{
    if (rad_module_register(handle, RAD_MODVERSION, &modinfo) == -1)
        return (-1);

    /* This module must be run as the authenticated user */
    if (rad_isproxy)
        return (0);

    (void) cont_insert_singleton(rad_container, adr_name_fromstr(
        "com.oracle.solaris.user:type=UserManager"),
        &interface_UserMgr_svr);

    return (0);
}
```

◆ ◆ ◆ CHAPTER 7

rad Best Practices

This chapter provides guidance when using `rad`. The guidance material is grouped around the following topics.

- When to use `rad`?
- How to use `rad`?

When To Use rad?

`rad` is designed to provide remote administrative interfaces for operating system components/sub-systems. Such interfaces support the distributed administration of systems and greatly increase the abilities of system administrators to support large installations.

It is not intended to be a general purpose mechanism for building distributed applications, many alternative facilities, for example, RPC, RMI, CORBA, and MPI exist for such applications.

How To Use rad?

This section contains specific guidance on how to use `rad`.

API Guidelines

Designing a `rad` API requires judgement and the application of domain knowledge.

Target Audience

The users of the API fall into two broad categories:

- Administrators
- Developers

Unfortunately, accommodating the desires of consumers in these two categories within one interface is difficult. The first group desire task-based APIs which match directly onto well-understood and defined administrative activities. The second group desire detailed, operation-based interfaces which may be aggregated to better support unusual or niche administrative activities.

For any given subsystem, you can view existing command-line utilities (CLIs) and libraries (APIs) as expressions of the rad APIs which are required. The CLIs represent the task-based administrative interfaces and the APIs represent the operation-based developer interfaces.

The goal in using rad is to provide interfaces that address the lowest-level objectives of the target audience. If targeting administrators (task-based), this effort could translate to matching existing CLIs. If targeting developers, this effort could mean significantly less aggregation of the lower-level APIs.

Legacy Constraints

Many subsystems present incomplete interfaces to the world. Some CLIs contain processing capabilities that are not accessible from an existing API. This situation is another motivation for providing task-based administrative interfaces before introducing more detailed interfaces.

Such constraints must be considered in the rad API design. Consider migrating functionality from the CLI into the API to facilitate the creation of the new interface. Also consider presenting an interface which wraps the CLI and takes advantage of the existing functionality. Do not simply duplicate the functionality in the new rad interface, which would introduce redundancy and significantly increase maintenance complexity. One particular area where rad interface developers need to be careful is to avoid duplication around parameter checking and transformation. This duplication is likely to be a sign that existing CLI functionality should be migrated to an API.

rad modules must be written in C. Some subsystems, for instance, those written in other languages, have no mechanism for a C module to access API functionality. In these cases, rad module creators must access whatever functionality is available in the CLI or make a potentially significant engineering effort to access the existing functionality, for example, rewriting existing code in C, embedding a language interpreter in their C module, and the like.

Conservative Design

Designing a rad interface is very similar to designing a library interface. The same general principles of design apply: be conservative, start small, consider evolutionary paths and carefully consider commitment levels.

Once an interface is established, the use of versioning and considered, incremental improvements will expand the functionality.

Component Guidelines

This section presents specific design advice on the most significant components of a rad module. Naming is addressed separately in "Naming Guidelines" on page 70

API Guidelines

APIs are the primary deliverable of a rad module. They are a grouping of interfaces, events, methods and properties which enable a user to interact with a subsystem.

When exposing the elements of a subsystem consider carefully how existing functions can be grouped together to form an interface. Imperative languages, such as C, tend to pass structures as the first argument to functions, which provides a clear indicator as to how best to group functions into APIs.

Method Guidelines

Methods provide mechanisms for examining and modifying administrative state.

Consider grouping together existing native APIs into aggregated rad functions which enable higher order operations to be exposed.

Follow established good practice for RPC style development. rad is primarily for remote administration, and avoiding excessive network load is good practice.

Property Guideline

Make sure to define an <error> element with properties which can be modified.

Event Guidelines

The module is responsible for providing a sequence number. Monotonically increasing sequence numbers are recommended for use, since these will be of most potential use to any clients.

Consider providing mechanisms for allowing a client to throttle event generation.

Carefully design event payloads to minimize network load.

Don't try to replicate the functionality of network monitoring protocols such as SNMP.

Module Location: Deciding between Proxy or Slave

Judicious use of the is_proxy variable enables you to control where a module is loaded for execution: in the rad proxy or in a slave process.

A module should, by default, be loaded into a slave process unless the following conditions apply:

- Module performs self-authentication
- Module is very simple and cannot fail fatally

Synchronous and Asynchronous Invocation

All method invocations in rad are synchronous. Asynchronous behavior can be obtained by adopting a design pattern that relies on the use of events to provide notifications. Refer to "Synchronization" on page 61 for more details.

Duplication

Do not duplicate code from existing CLIs. Instead, consider moving common code into a lower library layer that can be shared by rad and the CLI.

Client Library Support

rad modules are designed to have a language agnostic interface. However, you might want to provide additional language support through the delivery of a language-specific extension. This type of deliverables should be restricted in use. The main reason for their existence is to help improve the fit of an interface into a language idiom.

Naming Guidelines

When naming an API, interface, or object, module developers have broad leeway to choose names that make sense for their modules. However, some conventions can help avoid pitfalls that might arise when retrieving objects from the rad server.

Object Names

The domain portion of rad object names follows a reverse-dotted naming convention that prevents collisions in rad's flat object namespace. This convention typically resembles a Java package naming scheme:

```
com.oracle.solaris.rad.zfs
com.oracle.solaris.rad.zonesmgt
com.oracle.solaris.rad.usermgt
org.opensolaris.os.rad.ips
...
```

To distinguish a rad API from a native API designed and implemented for a specific language, include a "rad." component in the API name.

With the goal of storing objects with names consumers would expect, APIs, and the domains of the objects defined within them, should share the same name. This practice makes the mapping between the two easily identifiable by both the module consumer and module developer.

With the same goal of simplicity, identifying an interface object is made easier by adhering to a "type=interface" convention within the object name.

Applying both conventions, a typical API will look like the following example.

```
<api
  xmlns="http://xmlns.oracle.com/radadr"
  name="com.oracle.solaris.rad.zfs">

  <interface name="ZPool">
    <summary>
      zpool administration
    </summary>
    ...
```

Within the module, the API appears as follows:

```
int
_rad_init(void *handle)
{
    ...
    adr_name_t *name =
        adr_name_fromstr("com.oracle.solaris.rad.zfs:type=ZPool");
    (void) cont_insert_singleton(rad_container, name, &interface_ZPool_svr);
```

On the consumer side (Python), the API appears as follows:

```
import rad.client
import rad.util

# Create a connection
radconn = rad.util.connect_unix()

# Retrieve a ZPool object
zpool_name = rad.client.Name("com.oracle.solaris.rad.zfs",
    [("type", "ZPool")])
zpool = radconn.get_object(zpool_name)
```

Using this naming convention also precludes the need to specify a pragma to identify a package when generating Java interfaces because `radadrgen`(1) uses the API name as the Java package name by default.

Case

In an effort to normalize the appearance of like items across development boundaries, and to minimize the awkwardness in generated language-specific interfaces, several case strategies have been informally adopted.

Module	The base of the API/domain name. For a module describing an interface *domain.prefix.base*.`xml`, module spec files should be named *base*.`xml`, and the resulting shared library `mod_`*base*.`so`. Examples: ■ `/usr/lib/rad/apis/zfs.xml` ■ `/usr/lib/rad/module/mod_zfs.so`

API	Reverse-dotted domain, all lowercase. Examples: ■ `com.oracle.solaris.rad.zfs` ■ `com.oracle.solaris.rad.zonemgt`
Interface, struct, union, enum	Non-qualified, camel case, starting with uppercase. Examples: ■ `Time` ■ `NameService` ■ `LDAPConfig` ■ `ErrorCode`
Enum value and fallback	Non-qualified, uppercase, underscores. Examples: ■ `CHAR` ■ `INVALID_TOKEN` ■ `REQUIRE_ALL`
Interface property and method, struct field, event	Non-qualified, camel case, starting with lowercase. Examples: ■ `count` ■ `addHostName` ■ `deleteUser`

Language-Specific Considerations

Though ADR is language-neutral, certain environments might have conventions that place additional constraints on interface design.

Currently known language-specific constraints:

- JMX: Interface-defined method names that resemble derived method names

A JMX `MXBean` Proxy has a single namespace for both methods defined by the interface, and derived methods used for accessing attributes defined by the interface. If a method defined by the interface has a name and signature that is consistent with the JavaBeans-style naming of a derived attribute-access method, then the Proxy will assume calls to it are attempts to access a foo attribute on the JMX object and will fail. For example:

```
public void setFoo(String s);
public int getFoo();
public boolean isFoo();
```

This constraint is not a limitation of Java or of JMX but of the Proxy implementation. The designer could choose not use the Proxy, or to use a Proxy implementation that does not have this limitation.

API Design Examples

Combining the tools described so far in this document to construct an API with a known design can be a challenge. Several possible solutions for a particular problem are often available. The examples in this section illustrate the best practices described in previous sections.

User Management Example

Object/interface granularity is subjective. For example, imagine an interface for managing a user. The user has a few modifiable properties:

TABLE 7–1 Example User Properties

Property	Type
name	string
shell	string
admin	boolean

The interface for managing this user might consist solely of a set of attributes corresponding to the above properties. Alternatively, it could consist of a single attribute that is a structure containing fields that correspond to the properties, possibly more efficient if all properties are usually read or written together. The object implementing this might be named as follows:

```
com.example.users:type=TheOnlyUser
```

If instead of managing a single user you need to manage multiple users, you have a couple of choices. One option would be to modify the interface to use methods instead of attributes, and to add a "user" argument to the methods, for example:

```
setUserAttributes(username, attributes) throws UserError
attributes getUserAttributes(username) throws UserError
```

This example is sufficient for a single user, and provides support to other global operations such as adding a user, deleting a user, getting a list of users and so on. You might want to give it a more appropriate name, for example:

```
com.example.users:type=UserManagement
```

However, suppose there were many more properties associated with the user and many more operations you would want to do with a user, for example, sending them email, giving them a bonus and so on. As the server functionality grows, the UserManagement's API grows increasingly cluttered. It would accumulate a mixture of global operation and per-user operations, and the need for each per-user operation to specify a user to operate on, and specify the errors associated with not finding that user, would start looking redundant.

```
username[] listUsers()
addUser(username, attributes)
giveRaise(username, dollars) throws UserError
fire(username) throws UserError
sendEmail(username, message) throws UserError
setUserAttributes(username, attributes) throws UserError
attributes getUserAttributes(username) throws UserError
```

A cleaner alternative would be to separate the global operations from the user-specific operations and create two interfaces. The UserManagement object would use the global operations interface:

```
username[] listUsers()
addUser(username, attributes)
```

A separate object for each user would implement the user-specific interface:

```
setAttributes(attributes)
attributes getAttributes()
giveRaise(dollars)
fire()
sendEmail(message)
```

Note that if fire operates more on the namespace than the user, it should be present in UserManagement where it would need to take a username argument.

Finally, the different objects would be named such that the different objects could be easily differentiated and be directly accessed by the client:

```
com.example.users:type=UserManagement
com.example.users:type=User,name=ONeill
com.example.users:type=User,name=Sheppard
...
```

This example also highlights a situation where the rad server may not want to enumerate all objects when a client issues a `LIST` request. Listing all users may not be particularly expensive, but pulling down a list of potentially thousands of objects on every `LIST` call will not benefit the majority of clients.

◆ ◆ ◆ APPENDIX A

rad Binary Protocol

In addition to supporting multiple transports, `rad` is capable of talking different protocols. The default and sole protocol is a proprietary binary protocol designated `rad`. This appendix documents version 1 of this protocol.

Overview

The `rad` protocol is a bidirectional binary protocol that operates over a single stream. Communication in both directions takes the form of discrete messages. These messages are framed using RPC record marking. For more information, see the "RECORD MARKING STANDARD" section in the RPC standard.

The individual messages take the formats documented in this appendix. Even though RPC record marking permits skipping messages of unknown format, both the client and the server are free to immediately drop the connection when an invalid message is seen.

The `rad` protocol is built using XDR, so for simplicity and clarity, the XDR primitive type names and syntax are used throughout this appendix. For example:

- "`FOO<>`" — represents a variable-length array of FOOs, communicated as an unsigned int containing the size followed by that number of FOOs.
- "`FOO[n]`" — represents a fixed-length array of a predetermined size, communicated only as the nFOOs.
- "`FOO *`" — represents an optional value communicated as a boolean value followed by a FOO if and only if the boolean value is true.

Common Data Formats

The types and concepts that appear repeatedly throughout the rad protocol are described in this section.

Operations

The various operations one can perform against the server are communicated with an operation code.

TABLE A–1 OP-CODE Description

Field Name	Length	Type	Description
op_code	4	int	operation code: 0 OC-INVOKE 1 OC-GETATTR 2 OC-SETATTR 3 OC-LOOKUP 4 OC-DEFINE 5 OC-LIST 6 OC-SUB 7 OC-UNSUB

Errors

Errors in the rad protocol are communicated by an error code, and optionally structured error data.

TABLE A–2 ERROR-CODE Description

Field Name	Length	Type	Description

`error_code`	4	`int`	error code:
			`0`
			`EC-OK`
			`1`
			`EC-OBJECT`
			`2`
			`EC-NOMEM`
			`3`
			`EC-NOTFOUND`
			`4`
			`EC-PRIV`
			`5`
			`EC-SYSTEM`
			`6`
			`EC-EXISTS`
			`7`
			`EC-MISMATCH`
			`8`
			`EC-ILLEGAL`

For object-specific errors, `EC-OBJECT`, the format of the structured data is defined by the object's interface definition. For the remainder, save for `EC-OK` (which indicates success), the format is defined during the initial connection handshake. See "Connection Initialization" on page 88.

Time

Times are represented in the `rad` protocol, as seconds and nanoseconds offset from the epoch (January 1, 1970 UTC).

Note that the accuracy of such time data is determined by its context. In nearly all cases, the full nanosecond precision is only relevant when compared to time data obtained from the same host.

TABLE A–3 TIME-DATA

Field Name	Length	Type	Description
`secs`	8	`hyper`	Number of seconds
`nsecs`	4	`int`	Number of nanoseconds ($0 \leq nsec \leq 10^9$)

Object Names

`rad` object names are structured, consisting of a domain and one or more key-value pairs. When an object name or pattern needs to be represented in the `rad` protocol, this structure is flattened to a canonical string format. This string format consists of the domain, followed by a colon (':'), followed by comma-separated key-value pairs. Each key-value pair consists of the name, followed by an equals sign ('='), followed by the value.

Because keys and values may contain the special characters '=' or ',', a quoting algorithm is applied when constructing the string. The substitutions listed in the following table are applied to keys and values.

TABLE A–4 Object Name Escaping

Plain Text	Escaped Text
\	\S
,	\C
=	\E

Backslashes are found only as part of quoted characters, and commas and equals signs, when present, always have their special meaning. Object names that use no special characters are passed through unchanged. These facts can be used to simplify parsing or preprocessing of string-formatted object names.

TABLE A–5 NAME-DATA

Field Name	Length	Type	Description
`name`	variable	`string<>`	Flattened object name

For example, an object name in the domain `com.example` with the following keys and values:

Key	Value
directory	C:\
first,last	Doe,John

would be represented by the string `com.example:directory=C:\S,first\Clast=Doe\CJohn`.

ADR Data

Central to the `rad` protocol is the communication of data defined by ADR. Most ADR primitives map directly to XDR types:

TABLE A–6 Primitive ADR to Wire Type Mapping

ADR Type	Wire Type
boolean	XDR `boolean`
integer	XDR `int`
uinteger	XDR `unsigned int`
long	XDR `hyper`
ulong	XDR `unsigned hyper`
float	XDR `float`
double	XDR `double`
string	XDR `string<>`
opaque	XDR `opaque<>`
secret	XDR `string<>`
time	`TIME-DATA` (see "Time" on page 77)
name	`NAME-DATA` (see "Object Names" on page 78)

Optional ADR data of any type is represented as XDR optional data.

TABLE A–7 OPTIONAL-DATA Description

Field Name	Length	Type	Description
`data`	Varies	`ADR-DATA*`	Optional encoded data

Array data is communicated as an XDR array of the element data represented by an XDR `unsigned int` whose value is the number of array elements followed by that number of element data.

TABLE A–8 ARRAY-DATA Description

Field Name	Length	Type	Description
`elements`	Varies	`ADR-DATA<>`	Array elements

Structure data is communicated by communicating each structure field in the order they are defined. This may consist of a mixture of nullable and non-nullable data.

TABLE A–9 STRUCT-DATA

Field Name	Length	Type	Description
`fields`	Varies	`ADR-DATA`[n]	Fixed-length array of structure fields (n = type-defined field count)

Enumeration data is communicated as an XDR unsigned int whose value is the 1-based index into the list of enumerated values, that is, 1 would be the first enumerated value, *n* would be the *n*th enumerated value. A value of `0` represents the fallback value. For enumerations without a fallback value, the 0 value is unused.

TABLE A–10 ENUM-DATA Description

Field Name	Length	Type	Description
`value`	4	`unsigned int`	0 if fallback, 1-based index otherwise

Union data is communicated as an XDR unsigned int whose value is the 1-based index into the list of union arms. A value of 0 represents the default arm. When a non-default arm is selected, the arm is followed by that arm's data. When the default arm is selected, it is first followed by the discriminant data value and then is followed by the default arm's data.

TABLE A–11 UNION-DATA (Non-Default) Description

Field Name	Length	Type	Description
`arm_index`	4	`unsigned int`	1-based index into list of arms
`arm_data`	Varies	`ADR-DATA`	Arm data

TABLE A–12 UNION-DATA (Default) Description

Field Name	Length	Type	Description
`arm_index`	4	`unsigned int`	0
`discriminant`	4	`ENUM-DATA` | `boolean`	Discriminant value
`arm_data`	Varies	`ADR-DATA`	Arm data

ADR types

As discussed in "ADR Data" on page 79, the ADR data communicated by the rad protocol has a structure determined by its type. Before that data can be communicated to the client, however, the types themselves must be communicated.

For efficiency, type data is communicated using a system of type spaces and type references. A type space is an array of type definitions that is referenced by a protocol-defined set of consumers. A consumer referring to a type will use a type reference, which points to either a primitive type or to an element of the type space.

Both type references and type spaces refer to the various types by using an integral type code.

TABLE A–13 TYPE-CODE

Field Name	Length	Type	Description
type_code	4	int	Type code: 0 TC-VOID 1 TC-BOOLEAN 2 TC-INTEGER 3 TC-UINTEGER 4 TC-LONG 5 TC-ULONG 6 TC-FLOAT 7 TC-DOUBLE 8 TC-TIME 9 TC-STRING 10 TC-OPAQUE 11 TC-SECRET 12 TC-NAME 13 TC-ENUM 14 TC-ARRAY 15 TC-STRUCT 16 TC-UNION

A type reference consists of an XDR `int` whose value is one of the type constants listed in the table. If the type is an `enum`, `array`, `union`, or `struct`, the type reference also includes an XDR `int` whose value is an index into the current type space as listed in the following tables.

TABLE A–14 TYPEREF (Basic Type)

Field Name	Length	Type	Description
type_code	4	TYPE-CODE	A primitive type code

TABLE A–15 TYPEREF (Derived Type)

Field Name	Length	Type	Description
type_code	4	TYPE-CODE	TC-ENUM, TC-ARRAY, TC-STRUCT, or TC-UNION
type_index	4	int	Type space index

A type space is a topologically sorted array of type definitions. A derived type may reference only derived types defined earlier in the type space. That is, a type reference used by the type at index X may reference only a primitive type or a derived type at index less than X. Recursively defined types are not supported.

Apart from a common distinguishing type code, each derived type is defined differently. Arrays are the simplest. An array definition consists of only a reference to the element type.

TABLE A–16 ARRAY-TYPE

Field Name	Length	Type	Description
type_code	4	TYPE-CODE	TC-ARRAY
element_type	Varies	TYPEREF	The element type

A structure type definition consists of a name and an array of field definitions. The order in which the fields are specified in STRUCT-TYPE is the order used to serialize the fields in STRUCT-DATA.

TABLE A–17 FIELD-TYPE

Field Name	Length	Type	Description
name	Varies	string<>	The field name
nullable	4	boolean	Is the field value nullable?
type	Varies	TYPEREF	The field type

TABLE A–18 STRUCT-TYPE

Field Name	Length	Type	Description
type_code	4	TYPE-CODE	TC-STRUCT
name	Varies	string<>	The structure name
fields	Varies	FIELD-TYPE<>	Ordered list of structure fields

A union type definition consists of a name, a reference to the discriminant type, optionally a default arm specification, and an array of arm definitions. The arm index referenced by UNION-DATA is an index into the array of arms defined by the corresponding UNION-TYPE.

TABLE A–19 ARM-TYPE

Field Name	Length	Type	Description
value	4	ENUM-DATA | boolean	The discriminant value that selects this arm
nullable	4	boolean	Is the arm's value nullable?
type	Varies	TYPEREF	The arm's type

TABLE A–20 UNION-TYPE (Without Default Arm)

Field Name	Length	Type	Description
type_code	4	TYPE-CODE	TC-UNION
name	Varies	string<>	The union name
disc_type	Varies	TYPEREF	The discriminant type
hasdefault	4	boolean	False
arms	Varies	ARM-TYPE<>	Ordered list of arms

TABLE A–21 UNION-TYPE (With Default Arm)

Field Name	Length	Type	Description
type_code	4	TYPE-CODE	TC-UNION
name	Varies	string<>	The union name
disc_type	Varies	TYPEREF	The discriminant type
hasdefault	4	boolean	True
def_nullable	4	boolean	Is the default arm value nullable?
def_type	Varies	TYPEREF	The default arm type

TABLE A–21 UNION-TYPE (With Default Arm) *(Continued)*

Field Name	Length	Type	Description
`arms`	Varies	`ARM-TYPE<>`	Ordered list of arms

Lastly, an `enum` type definition consists of a name, an optional fallback value, and a list of enumeration values. The value index referenced by `ENUM-DATA` is an index into the array of values defined by the corresponding `ENUM-TYPE`.

TABLE A–22 VALUE-TYPE

Field Name	Length	Type	Description
`name`	Varies	`string<>`	The value's name
`value`	4	`int`	The value's assigned value

TABLE A–23 ENUM-TYPE

Field Name	Length	Type	Description
`type_code`	4	`TYPE-CODE`	TC-ENUM
`name`	Varies	`string<>`	The enum name
`fb_name`	Varies	`string<>*`	The fallback value's name, if one exists
`values`	Varies	`VALUE-TYPE<>`	Ordered list of values

The type space itself is an array. Each element is one of these four type definitions (`ARRAY-TYPE`, `STRUCT-TYPE`, `UNION-TYPE`, or `ENUM-TYPE`).

TABLE A–24 TYPESPACE

Field Name	Length	Type	Description
`types`	Varies	`?-TYPE<>`	Ordered list of types in the type space

Interface Definitions

The ultimate description of the interactions permitted with a particular object is its interface definition. An interface definition contains many elements: an API name, a list of versioned interface names the interface supports, and definitions of the interface's attributes, methods, and events.

Each interface name has a set of stabilities and versions.

TABLE A–25 STABILITY-CODE

Field Name	Length	Type	Description
stability_code	4	int	Stability code: 1 SC-PRIVATE 2 SC-UNCOMMITTED 3 SC-COMMITTED

TABLE A–26 VERSION-DATA

Field Name	Length	Type	Description
stability	4	STABILITY-CODE	Stability version applies to
major	4	int	Major version number
minor	4	int	Minor version number

TABLE A–27 INTERFACENAME-DATA

Field Name	Length	Type	Description
interface_name	Varies	string<>	Interface name
versions	Varies	VERSION-DATA<>	Interface versions by stability

An attribute consists of a name, stability, various flags, a type, and separate, optional read and write error types.

TABLE A–28 ATTRIBUTE-TYPE

Field Name	Length	Type	Description
aname	Varies	string<>	Attribute name
stability	4	STABILITY-CODE	Stability
readable	4	boolean	Is attribute readable?
writable	4	boolean	Is attribute writable?
nullable	4	boolean	Is attribute nullable?
type	Varies	TYPEREF	Attribute type
read_error	Varies	TYPEREF *	Error data on read, if applicable

TABLE A–28 ATTRIBUTE-TYPE *(Continued)*

Field Name	Length	Type	Description
`write_error`	Varies	`TYPEREF *`	Error data on write, if applicable

A method resembles an attribute. It has a name, stability, a result type, only a single optional error type, and an array of argument definitions.

TABLE A–29 ARGUMENT-TYPE

Field Name	Length	Type	Description
`argname`	Varies	`string<>`	Argument name
`nullable`	4	`boolean`	Is argument nullable?
`type`	Varies	`TYPEREF`	Argument type

TABLE A–30 METHOD-TYPE

Field Name	Length	Type	Description
`mname`	Varies	`string<>`	Method name
`stability`	4	`STABILITY-CODE`	Stability
`nullable`	4	`boolean`	Is result nullable?
`result_type`	Varies	`TYPEREF`	Result type
`error`	Varies	`TYPEREF *`	Error data, if applicable
`args`	Varies	`ARGUMENT-TYPE<>`	Arguments

An event consists only of a name, stability, and type.

TABLE A–31 EVENT-TYPE

Field Name	Length	Type	Description
`ename`	Varies	`string<>`	Method name
`stability`	4	`STABILITY-CODE`	Stability
`event_type`	Varies	`TYPEREF`	Event type

An interface definition combines all of the described elements.

TABLE A–32 INTERFACE-TYPE

Field Name	Length	Type	Description
api_name	Varies	string<>	API name (domain)
interfaces	Varies	INTERFACENAME-DATA<>	Interface names and versions implemented
types	Varies	TYPESET<>	Types used by interface definition
attributes	Varies	ATTRIBUTE-TYPE<>	Interface attributes
methods	Varies	METHOD-TYPE<>	Interface methods
events	Varies	EVENT-TYPE<>	Interface events

Connection Initialization

Once a connection has been established between a client and server, a short synchronous handshake initiates the rad protocol. The server begins by sending a SERVER-HELLO message. This message specifies the minimum and maximum protocol versions (inclusive) recognized by the server.

TABLE A–33 SERVER-HELLO

Field Name	Length	Type	Description
protocol	3	string[3]	“RAD”
min_ver	4	int	Minimum supported version
max_ver	4	int	Maximum supported version

The client then replies with a CLIENT-HELLO message specifying the version it wishes to use. This version may not be less than the server's minimum version or greater than the server's maximum version.

TABLE A–34 CLIENT-HELLO

Field Name	Length	Type	Description
protocol	3	string[3]	“RAD”
version	4	int	Client-selected version
locale	Varies	string<256>	Client locale

Part of the rad protocol is the communication of structured error data on request failures. For consistency with the object-specific errors that server-side objects are permitted to return, the

errors returned by rad when requests fail for other reasons are also defined using ADR. After the server receives and accepts a CLIENT-HELLO message, it replies with ERRORS to communicate those type definitions.

TABLE A–35 ERRORS

Field Name	Length	Type	Description
error_space	Varies	TYPESPACE	Typespace containing error types
errors	Varies	TYPEREF<>	Error types, starting with EC-NOMEM

The TYPEREFs refer to the types defined in the error_space TYPESPACE. The types define the first error_count errors, starting from the first non object-specific error, EC-NOMEM. Any errors for which types are not defined have type void.

At this point, the handshake is complete and normal client-server communication can occur.

Messages

Normal communication consists of an asynchronous exchange of messages: REQUESTs from the client to the server, RESPONSEs from the server to the client, and EVENTs from the server to the client.

A REQUEST is the rad equivalent of a low-level remote procedure call. It consists of a client-selected, non-zero serial number, an operation code, and an opaque, operation-specific, variable-length payload.

TABLE A–36 REQUEST

Field Name	Length	Type	Description
serial	8	hyper	Client-specified serial number
opcode	4	OP-CODE	Operation code (see "Operations" on page 76)
payload	Varies	opaque<>	Request payload

The server will respond to every REQUEST with a RESPONSE. A RESPONSE consists of the serial number of the corresponding REQUEST, an error code, and a response payload. If REQUEST succeeded, the error code will be EC-OK and the payload will contain the operation-defined payload data. Otherwise, the error code will reflect the type of failure and the payload will contain either the protocol-defined or object-defined error payload data.

TABLE A–37 RESPONSE

Field Name	Length	Type	Description
`serial`	8	`hyper`	Serial number of REQUEST
`error`	4	`ERROR-CODE`	Error code (see "Errors" on page 76)
`payload`	Varies	`opaque<>`	Response payload

The `rad` protocol does not require the client to wait for a `RESPONSE` before sending another `REQUEST`. However, the server implementation might place limits on the number of outstanding requests that can be handled simultaneously. A client with an outstanding `REQUEST` must assume that a subsequent `REQUEST` will block until the `RESPONSE` from the outstanding `REQUEST` is read.

A client may (through a `REQUEST`) subscribe to asynchronous event sources. When an event occurs, the server will send an `EVENT` message to the client. An `EVENT` message will include the object ID of the source object, the time of the event, the name of the event, and an event-specific opaque payload. An `EVENT` message is distinguished from a `RESPONSE` by having a serial number of `0`.

TABLE A–38 EVENT

Field Name	Length	Type	Description
`serial`	8	`hyper`	0
`source`	8	`hyper`	ID of generating object
`sequence`	8	`hyper`	Sequence number of event
`timestamp`	12	`TIME-DATA`	Time of event
`name`	Varies	`string<>`	Event name
`payload`	Varies	`opaque<>`	Event payload

Operations

Each operation that a client can perform against a `rad` server has its own request and response payloads. To facilitate processing without needing to fully decode the payload, these payloads are communicated as variable-lengthed `opaque` data in the `REQUEST` and `RESPONSE`.

For consistency and flexibility, all `ADR` data referenced by these payloads is communicated as `OPTIONAL-DATA`, which in turn is wrapped as opaque data.

TABLE A–39 PAYLOAD-DATA

Field Name	Length	Type	Description
`data`	Varies	`opaque<>`	Encapsulated OPTIONAL-DATA

INVOKE Operation

`INVOKE` makes a method call against a `rad` object, identified by its object ID.

TABLE A–40 INVOKE-REQUEST

Field Name	Length	Type	Description
`objectid`	8	`hyper`	Object ID, returned by lookup
`mname`	Varies	`string<>`	The method to invoke
`arguments`	Varies	`PAYLOAD-DATA<>`	Array of method arguments

TABLE A–41 INVOKE-RESPONSE

Field Name	Length	Type	Description
`result`	Varies	`PAYLOAD-DATA`	The return value of method call, if any

`INVOKE` can fail for the following reasons:

EC-OBJECT The method call was made but failed for an object-specific reason.

EC-NOTFOUND objectid is not a known object ID or the object does not have the method mname.

EC-MISMATCH The wrong number of arguments were provided, or a non-nullable argument was missing.

EC-NOMEM The server had insufficient resources to complete the operation.

EC-SYSTEM An unexpected internal error occurred.

GETATTR Operation

`GETATTR` reads an attribute of a `rad` object identified by its object ID.

TABLE A–42 GETATTR-REQUEST

Field Name	Length	Type	Description
objectid	8	hyper	Object ID, returned by lookup
aname	Varies	string<>	The attribute to read

TABLE A–43 GETATTR-RESPONSE

Field Name	Length	Type	Description
result	Varies	PAYLOAD-DATA	The value of the attribute

GETATTR can fail for the following reasons:

EC-OBJECT — An attempt to read the attribute was made, but failed for an object-specific reason.

EC-NOTFOUND — objectid is not a known object ID or the object does not have the attribute aname.

EC-ILLEGAL — aname refers to a write-only attribute.

EC-NOMEM — The server had insufficient resources to complete the operation.

EC-SYSTEM — An unexpected internal error occurred.

SETATTR Operation

SETATTR reads an attribute of a rad object identified by its object ID. The response payload for a SETATTR request is empty.

TABLE A–44 SETATTR-REQUEST

Field Name	Length	Type	Description
objectid	8	hyper	Object ID, returned by lookup
aname	Varies	string<>	The attribute to write
value	Varies	PAYLOAD-DATA	The new value of the attribute

SETATTR can fail for the following reasons:

EC-OBJECT — An attempt to write the attribute was made, but failed for an object-specific reason.

EC-NOTFOUND	objectid is not a known object ID or the object does not have the attribute aname.
EC-MISMATCH	aname has a non-nullable value and `value` was `NULL`.
EC-ILLEGAL	aname refers to a read-only attribute.
EC-NOMEM	The server had insufficient resources to complete the operation.
EC-SYSTEM	An unexpected internal error occurred.

LOOKUP Operation

`LOOKUP` attempts to find the named object in the server's namespace, returning the object and interface IDs of the object if it exists. Because an object is not usable until its interface has been defined, the client may request the interface definition be provided as part of the `LOOKUP` response. For the same reason, the server may unilaterally decide to provide the interface definition if it believes the client has not seen it yet.

TABLE A–45 LOOKUP-REQUEST

Field Name	Length	Type	Description
`name`	Varies	`NAME-DATA`	Object name
`define`	4	`boolean`	Include object definition?

TABLE A–46 LOOKUP-RESPONSE

Field Name	Length	Type	Description
`objectid`	8	`hyper`	ID of the object
`interfaceid`	8	`hyper`	ID of the object's interface
`definition`	Varies	`INTERFACE-TYPE *`	The definition of the object's interface

`LOOKUP` can fail for the following reasons:

EC-NOTFOUND	name does not exist.
EC-NOMEM	The server had insufficient resources to complete the operation.
EC-SYSTEM	An unexpected internal error occurred.

DEFINE Operation

`DEFINE` requests a definition of the specified interface ID.

TABLE A–47 DEFINE-REQUEST

Field Name	Length	Type	Description
`interfaceid`	8	`hyper`	Interface ID

TABLE A–48 DEFINE-RESPONSE

Field Name	Length	Type	Description
`definition`	varies	`INTERFACE-TYPE`	The definition of the interface

`DEFINE` can fail for the following reasons:

EC-NOTFOUND interfaceid isn't a known interface ID.

EC-NOMEM The server had insufficient resources to complete the operation.

EC-SYSTEM An unexpected internal error occurred.

LIST Operation

`LIST` requests an enumeration of all objects present in the server that match the specified object name pattern. The empty string matches all server objects.

TABLE A–49 LIST-REQUEST

Field Name	Length	Type	Description
`pattern`	Varies	`NAME-DATA`	Object name pattern

TABLE A–50 LIST-RESPONSE

Field Name	Length	Type	Description
`names`	Varies	`NAME-DATA<>`	The definition of the interface

`LIST` can fail for the following reasons:

EC-NOMEM The server had insufficient resources to complete the operation.

EC-SYSTEM An unexpected internal error occurred.

.

SUB and UNSUB Operations

SUB and UNSUB subscribe and unsubscribe, respectively, to the named event of the specified object. The response payload for a successful SUB or UNSUB is empty.

Note that it is possible to receive an EVENT that has been unsubscribed even after a successful UNSUB operation.

TABLE A–51 SUB-REQUEST

Field Name	Length	Type	Description
objectid	8	hyper	ID of the object
event	Varies	string<>	Event name

TABLE A–52 UNSUB-RESPONSE

Field Name	Length	Type	Description
objectid	8	hyper	ID of the object
event	Varies	string<>	Event name

SUB can fail for the following reasons:

EC-NOTFOUND objectid is not a known object ID or the object does not have the event event.

EC-EXISTS The client is already subscribed to event.

EC-NOMEM The server had insufficient resources to complete the operation.

EC-SYSTEM An unexpected internal error occurred.

UNSUB can fail for the following reasons:

EC-NOTFOUND objectid is not a known object ID the object does not have the event event, or the client isn't subscribed to event.

EC-NOMEM The server had insufficient resources to complete the operation.

EC-SYSTEM An unexpected internal error occurred.