Triton Reference Manual

Version 0.7.3

Dennis Heimbigner

CU-CS-483-91 Revised 31 Jan. 1991

University of Colorado at Boulder

Technical Report CU-CS-483-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Triton Reference Manual

Version 0.7.3

Dennis Heimbigner

Revised 31 Jan. 1991

1 Introduction

Triton is a program for providing access to persistent typed objects. It provides an interface by which other programs may dynamically create new types, new methods (in the behavioral object oriented sense), and new instances of the types. These instances are persistent, which means that they exist when the program that created them terminates, unless that program deliberately destroys the objects. These objects can also persist over instantiations of Triton, with each new instantiation having access to the objects that existed at the termination of the previous instantiation.

Triton is often referred to as an "object manager," but is more appropriately termed an "object manager *shell*." The term *shell* is used to indicate that while the Triton interface provides many of the services available through object managers, Triton itself is wrapped around an existing object manager, with the intent that Triton can provide some services not provided by the underlying object manager.

In this case, that object manager is Exodus, which comes from the University of Wisconsin. Exodus provides a low level storage manager to manage storage objects. A storage object is a contiguous sequence of bytes with an associated unique identity. These objects are kept on disk and cached in buffers in main memory as required. Exodus also provides a persistent programming language called E, and which is derived from C++. Thus the data model provided by E consists of the normal C type system (int, char, struct, array, etc.) plus *classes*, which encapsulate data and methods (procedures) that operate on that data. Classes may be arranged in a subclass tree and methods may be inherited down that tree. Multiple inheritance is not provided in this version of E, since it is based on C++ version 1.2.

E-Mail Address: dennis@cs.colorado.edu

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1012. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

2 Required Reading

This document assumes some significant knowledge about Exodus and Q. In particular, it assumes some knowledge about the basic structure of E, and it assumes that you have some understanding about remote procedure calls and how to use Q to encode and decode procedure arguments. In section 17, it is assumed that the reader is familiar with APPL/A.

3 Triton Architecture

An instance of Triton is a server. This means that it runs as a single Unix process and communicates with clients via remote procedure calls. The protocol used is the Sun RPC/XDR protocol over TCP/IP stream connections. Multiple clients may access the same server and their requests will be handled on a first-come, first-serve basis. At the moment, Exodus provides no transaction management facilities. However, a rudimentary form of atomicity is provided by the server in that each client request is executed to completion before any new request is processed.

A Triton server has some detailed interior structure (see Figure 1). At the lowest level is the exodus storage manager. On top of that there is the code for a collection of E class methods and functions. This code is intended to represent client application, and this code can be added dynamically to the Triton server as new applications come into being and use the server.

In addition to this application code, there is code to maintain a catalog of information about the client application classes, methods, and functions that are known to the server. There is also a small amount of code to support the remote procedure call interface.

4 Invoking Triton

In order to initiate a Triton server, a number of resources must be established. First, a database volume must be established as a file into which Triton will store persistent data. As a rule, every running server should have its own volume file in its own directory. There are two reasons for this:

1. At the moment, no two servers can correctly share a single volume.
2. The server, when running, will create server-specific files in the current working directory that is in force when it starts.

The volume file needs to be created and initialized by running the Exodus utility "formatvol" using a sequence of commands such as:

```
rm -f $EVOLUME
formatvol
8
8
8
8
```

where the sequence of 8's defines the various parameters for the disk file. There is nothing magic about 8. If the triton server crashes, then it is possible that the data volume is corrupted. If you

suspect that it is, then you will have to re-format the file and reload the schema and re-create the instances.

A number of environment variables must be defined at the time the server is started. Additionally, some variables may optionally be defined:

EPATH: This variable should point to the directory containing the Exodus system.

EVOLUME: This variable gives the name (possibly as an absolute path) of the file to use as the data volume for the server.

TRITONPATH: This variable should point to the directory containing the Triton system corresponding to the executing server. Usually this variable need not be specified since the default is constructed at system configuration time to be correct.

TRITONSEARCHPATH: This is a search path in the same format as the PATH variable (i.e., colon separated paths). It is used by the server to locate the object files specified during dynamic loading. If you do not set this variable, then you will generally need to specify absolute path names for the dynamic loader. A utility function is provided for adding to this searchpath. It has the following signature.

```
char* triton_searchpath_add(char* dir, int first=1);
```

This adds the directory specified by the string first argument to the current searchpath list. If first is one, then it is added at the front, else at the back. A corresponding Ada procedure is also available.

TRITONRPCPROTOCOL: This is a string with either the value "TCP", or "UDP" to indicate what protocol the server should use for its remote procedure call servicing.

TRITONRPCNUMBER: This is a string representing a small integer. It indicates what program number the server should use for its remote procedure call servicing. In previous versions of the server, it was impossible to have more than one server per host machine. Now, however, if each server is started with a different value of this environment variable, then multiple servers may co-exist on a single machine, subject to still using different data volumes in different directories.

Note that if the variables TRITONPRCNUMBER and TRITONRPCPROTOCOL have different definitions between the server and client, no communication will be possible. Section 10.8 on properties has some more information on other, possibly useful, environment variables.

Once the above resources are satisfactorily established, starting up Triton requires nothing more than invoking the Triton program. If the command line has arguments, then they are assumed to be the names of object files to be pre-loaded into the server. Once started, clients may begin to make calls to the server to perform any of the Triton interface functions. Orderly shutdown of Triton is best performed using the "tmcd" utility (see Section 16.2).

As a server starts up, it will check to see if a conflicting server already exists on the same machine. If one is detected, it will print a message, and ask if it should continue or abort. If it is told to continue, the previous server will be requested to shutdown.

5 Client Access To Triton

In order for a client program to properly access the server, it too must establish a number of resources. In particular, a number of environment variables may be defined at the time the client is started. All of these variables have defaults, but you may not want to use them.

TRITONHOST: The name of the machine containing the server with which the client is supposed to interact.

TRITONRPCPROTOCOL: This is a string with either the value "TCP", or "UDP" to indicate what protocol the client should use for its remote procedure call servicing.

TRITONRPCNUMBER: This is a string representing a small integer. It indicates what program number the client should use for its remote procedure call servicing.

Note that if the variables TRITONPRCNUMBER and TRITONRPCPROTOCOL have different definitions between the server and client, no communication will be possible. Section 10.8 on properties has some more information on other, possibly useful, environment variables.

6 The Triton Type System

The Triton type system was deliberately chosen to be subset of the E type model so that E programs could be used to provide behavioral components for the model. In retrospect, this may not have been such a wise idea due to the compleicity of the E type model. But given that choice, we can see that classes, methods, and functions are the basic schema elements that an application can define to Triton. Methods may be viewed as functions that are associated with specific classes, while functions are free-standing and not associated with any class.

Classes are of three basic kinds: base, generic, and derived. Unless otherwise specified, the term class will refer to a base class. A generic class is one that can be instantiated to produce a new, derived, class. For a detailed understanding of generic and derived classes, refer to the E reference manual.

Methods are functions associated with classes. The primary difference between a function and a method is that a method takes a pointer to an instance of the class as an implicit first parameter. This parameter can be referenced through the pseudo-variable "this," just as in C++. Functions are not associated with any class, and so their arguments are all explicit. Both methods and functions have typed, formal arguments, and typed return values (possibly void). The allowable types of arguments and return values is specified by a component of the Triton type model called the representational type system (RTS) The Triton RTS model is a subset of the E model, and consists of the following elements:

- The typical C scalar types (void, char, short, int, long),
- Floating point types (float and double),
- Pointers (void*),
- Null terminated strings (char*),
- Byte strings (char*),

- E Pointers (dbvoid*),
- QDR(=XDR) encoded values, [1]
- Uninterpreted values.

Ignoring issues of implementation, all of the above should be obvious except the last three: E pointers, QDR values, and uninterpreted values. Discussion of QDR and uninterpreted types will be deferred.

In E, some types are distinguished as potentially persistent. These types are named like the ordinary C++ types, but preceded by the letters "db": dbint, dbstruct, etc. Because instances of these types may potentially be made persistent, pointers to instances of them must be capable of referring to a very large disk space into which persistent objects are stored. In order to accommodate very large data spaces, E pointers to such objects are 16 bytes long as opposed to the 4 bytes of a normal C/C++ pointer. It is true that an E pointer could have been treated as a byte string, or a structure, but it is so common that it is useful to provide a separate type for it.

You may have noticed that there is no structure type in the above list. This is intentional and the reason will be discussed in a subsequent section.

7 Passing Values to Triton

The principal operation that a client requests of Triton is to invoke some function (or method) with some set of arguments. This presents some serious problems because the function is in one process, Triton, and the argument values are in the client process. So, it is necessary to *copy* the arguments from the client to Triton before executing the function.

This problem is not unique to Triton, of course, but is in fact inherent in any remote procedure call system. Triton uses the Q system to provide its remote procedure call and argument copy facilities.

In Q, as in RPC/XDR, the client arranges to copy arguments by providing a type specific procedure (a qdr procedure) that *linearizes* a given instance of an object. This means that the qdr procedure walks the object, including pointer following, and encodes the object into a linear string of bytes in some standardized form.

As a consequence of the way RPC is set up, every remote function must be defined to take one, read-only, argument, which is an input, and return a single value which must encode all returned values. In practice, this means that a multi-argument function must treat its arguments as if they were elements of a structure that is to be linearized. Attempts to provide read/write arguments are typically doomed since the inputs, being copies, will be discarded after the function returns.

Once the arguments for a function have been linearized, they may be sent to the server, Triton in this case. At the server, the linearized arguments must be *de-linearized* into an instance of the object. Note that this object is a *copy* of the original object, and in fact may not be identical if the linearize procedure chooses not to encode all information associated with the object on the client side. However, the qdr procedure was written by the application

[1] Q is a variant of the Sun XDR protocol, but it uses the same data structures, so QDR for our purposes is the same as XDR. The primary difference is that a QDR encoding must be preceded by the size of the encoded data.

programmer, and so it is assumed to linearize as much information as is needed on the server side.

Thus the basic operation in Triton consists of receiving a request that specifies a function and a linearization of the arguments to that function. Triton then de-linearizes the arguments, calls the function, takes any result from the function, linearizes that result, and returns the result to the client.

8 Special Argument Types

If triton recorded the complete structure of all argument types and all result types [2] then it could automatically linearize and de-linearize any instance of a type for whose structure it had a representation. Unfortunately, the E type system is very complicated, so the current version of Triton only records limited information about types.

In the current implementation, if a function is defined to have arguments only of types scalar, float, E pointer, null-terminated string, or byte string (fixed length, or variable), then Triton can automatically extract the arguments from the linearization and pass them to the function.

We can now see why structures are not currently supported. If the argument was a structure, and its size was known, but not its interior structure, then it could be de-linearized by treating it like a fixed length byte string. This decoding, however, would not be correct if the encoded fields of the encoded structure differ from the decoded fields. For example, in Q, the linearized form of a floating point number is not the same as the machine form for that number. Thus, a structure that contained a floating point number would not be correctly handled by Triton. Until Triton contains a more complete representation of the type structure, it was felt that structures should not be provided as it was too easy to make this mistake.

A more serious problem concerns pointers, and especially linked list structures. If a structure contains a pointer to some other object, then in the linearization, the pointed-to object probably should be linearized as well. On de-linearization, the pointer and the pointed-to object must be reconstructed. Triton does not record enough structure at the moment to do that correctly. Instead, Triton provides a couple of escapes to deal with this problem (as well as the opaque structure problem).

In Triton, it is possible to say that some argument of some function is of type "RLI_QDR," and to specify some other function whose purpose it is to de-linearize objects for that argument. Then, when Triton encounters an argument of this type, it invokes the second function, gives it the linearization, and expects in return a pointer to the de-linearized object. It then passes that pointer to the original function as the argument.

In some cases, it may be impossible to specify to Triton anything about the arguments of a function, a function with variable numbers of arguments, for example. In this case, termed RLI_ENCODED, the function is defined to take exactly one argument: the encoded linearization. It is the total responsibility of that function to decode the linearization into any needed arguments.

[2] Something we hope it will do in the not too distant future.

9 Special Result Types

As with argument types, the simpler result types for functions and methods can be specified and handled correctly by Triton. This includes scalars, floats, and E pointers.

If the result is a null terminated string, then it is assumed that the function returns a pointer to that string. Note that at the moment, there is no way to automatically free the space for the return string if it was dynamically allocated, and so you may have a memory leak in your system for this case.

If the result is a byte string, then it is assumed that, again, a pointer to it is returned. But, it is assumed that the byte string is exactly as long as the size associated with the type, so variable length byte strings cannot be returned this way. Again, there is no way to recover the space if the byte string was dynamically allocated.

A function result may be specified to be of type RLI_QDR, in which case the function is expected to return a pointer to an object to be linearized by the associated qdr function.

Similarly, a function may be designated as returning a result of type RLI_ENCODED. In this case, the function is expected to return a QDR_Handle as its result, and it is assumed that any returned result has been encoded into the buffer associated with the returned handle. Note that this differs from RLI_QDR in that the function must itself invoke any encoding function, whereas with RLI_QDR, Triton will invoke the encode function.

As an example, suppose that function "f" has been defined to Triton as returning a result of type RLI_ENCODED. Its code will look something like the following:

```
QDR_Handle f(...)
{
    <compute>
    <encode result into QDR_Handle x>
    return x;
}
```

The only question is: where does the QDR_Handle in x come from? There are two answers to this.

First, you can create it yourself:

```
QDR_Handle f(...)
{
    QDR_Handle x;
    <compute>
    x = qdr_create();
    <encode result into QDR_Handle x>
    return x;
}
```

This has the disadvantage that a new buffer will be created on every call to f, rather than reusing the same buffer. This can be alleviated by code such as this:

```
QDR_Handle f(...)
{
    static QDR_Handle x = NULL;
```

```
    <compute>
    if(x == NULL) x = qdr_create();
    <encode result into QDR_Handle x>
    return x;
}
```

As an alternative and a convenience, Triton calls such functions with an extra argument which is a QDR_Handle into which the function can encode its arguments. Such a function would look like this:

```
QDR_Handle f(...,QDR_Handle x) // note the extra argument
{
    <compute>
    <encode result into QDR_Handle x>
    return x;
}
```

10 The Triton Interface

The Triton interface is divided into several parts:

1. Triton initialization/finalization,
2. Schema definition operations,
3. Schema lookup operations.
4. Schema element destruction operations.
5. Instance manipulation operations,
6. Dynamic loading operations,
7. Dynamic unloading operations,
8. Property manipulation operations.

10.1 Initialization and Finalization

Assuming that a triton server is operating on host "X", a client application must first connect to that server before attempting to perform any operations. When the client is finished, it must disconnect from the server. This does not cause the server to terminate. Under rare circumstances, the client may need to terminate the server as well. These activities are handled by the following operations:

Initialize_triton: The argument to this function is the name of the host machine on which the Triton server resides or is null. If the argument is null, then the property "TritonHost" is used to determine the host machine on which the server is running. Normally, the null value should be used. This operation should be called once, and before any other operations.

Finalize_triton: This operation is intended to clean up on the client and to notify the server that the client will no longer be using the server (at least until another initialize is performed).

Shutdown_triton: This operation is intended to cause the server to clean up and exit. The caller should provide an exit code to be used by the server when it exits. Before Triton 0.7.2, this operation always complained about RPC timeout because the implementation was not completely clean.

10.2 Schema Definition Operations

The term schema is used here to mean the collection of types, classes, functions, and methods known to Triton. You should always refer to the actual files containing the interface when programming as this document may be somewhat out of date. The C/E/C++ interface is in the files $TRITONHOME/include/Triton.h, $TRITONHOME/include/TritonClient.h, $TRITONHOME/include/TritonUtil.h, and $TRITONHOME/include/TritonDestroy.h and the Ada interface is in the files $TRITONHOME/src/ada/triton.a, $TRITONHOME/src/ada/triton_util.a, and $TRITONHOME/src/ada/triton_relation.a. The term "$TRITONHOME" is intended to stand for the path to the top level directory containing the Triton system.

The primary interface operations include the following.

Create_Type: This function takes as its arguments an indicator of the type, the size, a name for the type (possibly null), and possibly a handle to a qdr function if the first argument is RLI_QDR. If the size is unknown, -1 should be specified. A handle (an E pointer) is returned as a result. This handle can be used in any other operation argument where a type definition is required.

Create_Class:

Create_Class_Generic:

Create_Class_Derived: These three operations take a name and a parent class reference, define a class to Triton and return a handle for referencing the newly created class. For non-derived classes, the class becomes a subtype of the parent class. For derived classes, the parent specifies a generic class from which the derived class was generated.

Create_Function: This returns a handle for referencing a function of a given name and with a given result type. If the result type is null, then the default is type "void."

Create_Method: This returns a handle for referencing a method of a given name and with a given result type and within a given class.

Create_Method_Inherited:

Create_Method_Derived: Each Method in E is associated with a class, but some methods are actually inherited from a parent class or are derived from some generic class. These operations are defined by specifying the operation (basemethod) from which they are inherited or derived, as well as the class with which the method is associated. Be careful; the parent argument here is the subclass (for inherited methods) or the derived class (for derived methods). It is not the parent class (the superclass or generic class). By default,

the new method will use the same name and result type as defined for the basemethod. These defaults can be overridden by the last two arguments to this operation.

Create_Formal: The arguments to a function or method are created using this operation. it takes a name for the formal, a function with which the formal is associated, and a type for that formal. A null formal type defaults to type "long." Note that at the moment, arguments do not inherit from supertypes or generics, so you will need to re-define the argument types for each function, inherited or not.

10.3 Schema Lookup Operations

Again, refer to $TRITONHOME/include/TritonClient.h to see the Triton interface operations for converting names to references. In the case of lookup_method, it is necessary to specify a reference to a class since methods of the same name may exist in different classes. Similarly, formal argument lookups must specify the function or method in which to look.

10.4 Destroying Schema Elements

Deleting a type from a database is a tricky problem. What about instances of that type? What about things that point to instances of that Type? General answers to these problems are difficult, and it is often claimed that types should never be deleted, but instead, versions of them should be created [3].

Triton does not yet provide versioning, but for purposes of debugging, it was felt essential to provide some means for destroying schema elements. The best way to use this feature is when you are creating schema elements and before any instances are created, or if you don't care about the instances. There is only a single operation:

Destroy_schema_object: This operation takes a handle to some schema element, presumably produced by a lookup operation or a create operation, and removes it from the schema. If a class is removed, then all of its subclasses, derived classes (if the deleted class is a generic), and all of its methods are also deleted. If a method or function is deleted, then all of its formal parameters are deleted. Note that this means that it is illegal to use the same formal parameter object in more than one method or function. Deletion of any of these elements will not cause any non-class types (defined using "create_type") to be deleted. Deleting a non-class type should not be done unless all references to it have already been destroyed. Suggestions are being taken as to the desired behavior when a type is deleted that still has references to it. One suggestion is to replace all references to the type with a reference to the type "void."

10.5 Instance Manipulation Operations

Once a schema is created, it must be possible to populate it with instances and then access and manipulate those instance. The file $TRITONHOME/include/TritonClient.h shows the interface operations for accomplishing those capabilities.

Create_Instance: The basic functionality of create_instance is to create an instance of a specified class. Dynamically created objects may be created as persistent if and only if they

[3] Versioning has its own problems.

are created as elements of a persistent "collection". In E, the class "collection" is generic and is typically instantiated with some class as its parameter. Collections can themselves be dynamically created as instances of other collections, but eventually, there must be persistent variable (declared using the "persistent" keyword) containing an instance of some collection. In E terms, the call

```
create_instance(cl,qargs,coll)
```

is loosely equivalent to the E expression

```
in (coll) new cl(qargs).
```

Normally, when an instance is created, it must be initialized by constructor method for the class. A constructor is a method with the same name as the class. The constructor may need arguments in order to initialize the instance. These arguments are encoded in the second argument to this operation.

Evaluate: This operation is used to invoke a method or a function in the Triton schema. It requires a handle to the function, the linearized arguments, a place to put the linearized result, and if it is a method, then an instance of the class of the method.

Destroying an element can be done by defining and evaluating (see Section 10.5) the destructor function on the class and evaluating it with the object to be destroyed as instance. At this point, the pointer to the instance is no longer meaningful, and attempts to use it will cause fatal errors.

10.6 Dynamic Loading Operations

As described above, the actual code for methods and functions must be dynamically loaded into the server. In order to do this, Triton keeps around a copy of the symbol table produced by the loader when Triton was loaded. As new object files are added to Triton, successive versions of this Triton symbol table file are produced. In addition, the E compiler (as modified for Triton) can output extra definition information (e.g, class sizes). This information should also be given to the server, and in fact is essential if the "create_instance" operation is used.

Figure 2 shows the correspondence between various schema elements and the object and definitions files. Compiling an E program produces an object file containing the actual code for a method or function. The definitions file contains Class size information.

Dynamic loading adds the code of the object file into the Triton server, and "binds" the code to the appropriate schema element: function or method. This is shown in Figure 3. Note that a sequence of dynamic loads is recorded in Triton as a "stack" of object files pushed on the stack in order of loading. This stack is termed the "registry stack." Entries in this stack are named by the the immediate file name of the load file or defines file associated with that registry. Thus, if the registry was created by loading "/moet/staff1/arcadia/point.o" then its name is "point.o."

There are several operations involving dynamic loading.

Bind_Initial_Symbols: It is possible to pre-load commonly used classes into the Triton server. In order for the methods of these classes to be accessible, this operation must be invoked.

It scans the current symbol table looking for values for methods and functions found in the catalog. If it is given an argument, then it assumes that it is a definitions file, and it searches that file for the sizes for any classes in its schema.

Load_and_Bind: This operation specified an object file containing the code for methods and functions to be dynamically loaded into Triton. It is assumed that appropriate creation operations have already been invoked to define the methods and functions being loaded. If a definitions file is available, it should also be specified. This has the side effect of creating a new symbol table file.

10.7 Dynamic Unloading

In order to debug a set of methods, it is necessary to load them into Triton, and then test them out. If an error is discovered, it would be nice to "unload" those methods and try again [4].

The "unload_thru" operation provides just such a capability. It takes as its argument the name of a previously loaded object file and, in effect, pops the registry stack until the named file has been removed. If no argument is specified, or the name is of length zero, the the whole stack is popped. This has several consequences:

1. All symbol tables produced at or before the load of the specified argument will be purged.
2. All code loaded at or after the specified file will be removed from Triton.
3. Any class sizes, method bindings, and function bindings will be invalidated in the schema.

10.8 Property Manipulation Operations

Beginning with Triton version 0.7.2, both clients and servers maintain named sets of "properties." The idea is to mimic the Unix environment capability in which it is possible to define (name,value) pairs for parameters that will influence the behavior of the client and/or server. It is possible to interrogate the server or client for the values of various properties. Additionally, some of those properties may be modified. All property values are passed as strings, even if the underlying type of the property is some other type.

10.8.1 Programmatic Interface

The primary interface for obtaining property values is the function

```
extern char* get_triton_property_value(char* propname);
```

This function takes the name of some property and returns a string which is an encoding of the value of that property. If there is no such property, or some other error occurs, then this function will return a null pointer.

It is also possible to modify properties using the function

```
extern int set_triton_property_value(char* propname, char* newvalue);
```

[4] Unfortunately, the most common effect of a bad methods is to crash the server, but there is little that can be done about that problem in the current compiler-based implementation.

This function takes the name of a property and a string encoding the new value to be assigned to that property. It will return a value greater than zero if it succeeded, and a value of zero or -1 if it failed.

At the moment the value string should be one of three kinds: integer, pathlist, or a string. An integer is a possibly signed sequence of digits. A path list is a sequence of strings encoded as a single string by using a colon (':') as a separator. This means, obviously, that the consituent strings may not contain colons. For example, the string "abc:def:g" is a sequence of three strings: "abc", "def", and "g". It should be obvious that this form was chosen deliberately to match the form used by environment variables such as PATH. Finally, a string is any other sequence of characters. The proper choice of format is determined by the type of the property being set (see tables 1 and 2).

In case a program needs to see the complete list of properties, the following function may be used.

```
extern char* get_triton_property_list();
```

This function returns a sequence of strings, separated by colons, where each string is the name of a known property.

The above functions manipulate properties in the server. A corresponding set of operations can be used to manipulate properties in the client.

```
extern char* get_client_property_value(char* propname);
extern int set_client_property_value(char* propname, char* newvalue);
extern char* get_client_property_list();
```

Some (but not all) of the properties in the client have the same name as some (but not all) properties in the server.

In both the client and the server, each property is associated with a global variable which has type "int" for integer values, "char*" for strings, or "char**" for path lists. Dynamically loaded code in the server, and any client code, can access these values.

10.8.2 Property Initialization

The properties will be initialized as part of a call to any of the following functions:

- initialize_triton
- get_client_property_value
- set_client_property_value
- get_client_property_list
- get_triton_property_value
- set_triton_property_value
- get_triton_property_list

At that point, any of the global variables will have a properly defined value and may be accessed. In most cases, the initial value of a property is taken first from some specified environment variables and second from some specified default.

10.8.3 Client Defined Properties

At the moment, the following properties are defined for the client with the following names and semantics. Case in the names is significant. Table 1 summarizes information about the client properties, while the list below defines their semantics.

ClientVersion: This is the version number for the client library code.

TritonRPCProtocol This specifies what protocol the client should use for its remote procedure calls. Acceptable values are "TCP","Tcp","tcp", "UDP","Udp","udp".

TritonRPCNumber This specifies what program number the client should use for its remote procedure calls.

TritonRPCVersion This specifies what version number the client should use for its remote procedure calls.

TritonHost This specifies the machine on which the server is assumed to be running.

TritonRPCTimeout This specifies how long (in seconds) the client will wait for the server to respond to its remote procedure calls.

ClientAPPLADebugLevel: This specifies an integer (encoded as a string) defining the level of debug output to be produced by all APPL/A relations on the client side.

10.8.4 Server Defined Properties

At the moment, the following properties are defined for the server with the following names and semantics. Case in the names is significant. Table 2 summarizes information about the server properties, while the list below defines their semantics.

TritonVersion: This is the version number for the server.

EPath: This string specifies the path for the E system. It is assumed that immediately below this directory is a directory named lib containing libE.a.

TritonPath: This specifies the path to the directory containing the Triton system. It is assumed that immediately below it is a bin directory containing, among other things, the triton executable from which the server was instantiated, and a lib directory.

TritonDebugLevel: This specifies an integer (encoded as a string) defining the level of debug output to be produced by the server.

APPLADebugLevel: This specifies an integer (encoded as a string) defining the level of debug output to be produced by all APPL/A relations.

TritonRPCProtocol This specifies what protocol the server should use for its remote procedure services. Acceptable values are "TCP","Tcp","tcp", "UDP","Udp","udp".

TritonRPCNumber This specifies what program number the server should use for its remote procedure services.

TritonRPCVersion This specifies what version number the server should use for its remote procedure services.

TritonHost This specifies the machine on which the server is running.

TritonSearchPath This is a path list specifying a search path to be used by the dynamic loader in trying to locate the object file to be loaded.

11 Triggers

Triton provides the ability to trigger on the invocation of methods or functions. This means that one can associate *trigger* functions with some specified *target* function or method and have those triggers be automatically invoked before and/or after the target function is executed. This is typically used to provide unobtrusive monitoring and to provide forward and backward inferencing, as in APPL/A. The interface is in $TRITONHOME/include/TritonClient.h.

Add_Trigger: This operation specifies a target function on which to trigger, an indication of when to invoke the trigger (before, or after or both), and the trigger function. It returns an integer identifier for referring to this trigger.

Remove_Trigger: This takes a target handle and a trigger identifier and disassociates the corresponding trigger from that target.

In theory, the trigger id determines the exact combination of when, trigger, and target. In order to avoid long searches of all targets looking for an id match, it is required to be specified in the context of a specific target, and this is why "remove_trigger" takes two arguments.

Any defined formal arguments for trigger functions are ignored when the trigger is invoked. Instead, all triggers are assumed to have the following signature

```
void trigger(
     QDR_Handle qargs,
     QDR_Handle qresult,
     dbvoid* instance,
     T_function_p target,
     trigger_id id,
     T_function_p trigger
    );
```

The "qargs" argument is the linearized input arguments for the target function, and the "qresult" argument is the linearized output from the function. This argument will be NULL in the case of a "before" trigger. If a method is the target, then the "instance" argument is the object to which the method was being applied. This will be NULL if the target is not a method. The next two arguments duplicate the arguments to "remove_trigger" so that a trigger can, if it wants, remove itself. The last argument is the handle for the trigger function. There are two things to note. First, since the arguments are linearized, the trigger function must know how to decode them if it wishes to look at the individual arguments. Second, it is possible for the trigger to change the inputs before they are passed to the target function. It is also possible to change the result from the target before returning the result to the client.

A utility function, "create_trigger_function_schema", can be used to define a schema element for a trigger function with the correct formal parameters. It takes as arguments a function name and, if it is supposed to be a method, the name of the parent class.

12 Constructing a Triton Client Program

In order for a client program to use Triton, several software pieces must be constructed. First, the client must have an E source file that defines the structure of the client data to be stored in Triton. For purposes of this discussion, we will use an example in which the client wishes to store points consisting of x and y coordinate pairs. Further, assume the client wishes to keep a persistent set of all points defined.

The file $TRITONHOME/src/tests/point.e contains the E code for the point class definition. It declares a class point with two fields: x and y. It has a constructor (method "point") and two other methods: "get_point" and "set_point." Since get_point must return two values, it is defined to return a value of type "struct xy".

After the definition of the point class in the file, there is a declaration of a point_collection, which is a class derived from the built-in generic class "collection." Then, a persistent instance of point_collection is declared (in the variable named "point_s." Finally, a function is defined whose sole purpose is to return a handle to the persistent set.

Once the client definition file has been defined, it should be compiled by the E compiler using a command of the form

```
EC +tpoint.d -I$TRITONDIR/include -I$QDIR -c point.e
```

The "+t" argument causes the compiler to place extra definition information into the file "point.d." [5] The "-I$TRITONDIR" denotes the fact that your program will generally include the triton header file "Triton.h.". The macro $TRITONDIR is supposed to represent the path to the directory containing Triton. Similarly, "$QDIR" points to the Q directory. It is needed since "Triton.h" depends on "QDR.h."

If the object file contains the code for a derived class, then it must be further processed by the "makeglobal" utility provided with Triton (see Section 16.1).

Now consider $TRITONHOME/src/tests/c1.e, which is an example of a client program. Generally, the client will invoke the following sequence of triton operations:

1. Initialize_triton(),
2. A sequence of schema definition operations (create_class, etc),
3. Load_and_bind(codefile,deffile), where for example, codefile is "point.o" and deffile is "point.d." [6]

At this point, the code is client dependent. The first function invoked is "get_point_space", whose handle has been placed in the variable "f_space." It takes no input argument, but its

[5] Caution: the "+t" argument should be the first argument to the EC command; other placement may or may not work.

[6] If the environment variable TRITONSEARCHPATH contains the path $TRITONHOME/src/tests, then the strings "point.o", and "point.d" can be used. Otherwise, the absolute paths "$TRITONHOME/src/tests/point.o" and "$TRITONHOME/src/tests/point.d" must be used.

result is an epointer handle to the persistent point collection. There is a bit of trickery here involving E pointers and normal pointers (see section 15). but the result is to leave the point collection handle in the variable "points."

Next, this client creates an instance of class point with initial value x=5 and y=18. To do this, the two arguments are linearized, and then create_instance is called to create and initialize the instance.

The next action of the program is to invoke "get_point" to retrieve the contents of the created point, and print them out.

Finally, "finalize_triton" is called to clean up, and the client exits.

If the client is executed again, then it will again attempt to create the schema and load "point.o." The server is defined so that redefining schema objects does not create duplicates in the schema. Further, object files that it believes to have been loaded correctly will not be reloaded. This has merits and demerits. It means that many clients can be run several times without problems. It means that debugging schemas and dynamically loaded objects files is difficult. One has to either deliberately restart the server (typically using "shutdown_triton") to remove the dynamically loaded code, or use the tcmd "unload" command (see section 16.2).

13 Caveats

Triton is frankly a fragile system, in that it can easily break, or get confused. This section describes some of the known problems in using Triton.

13.1 RPC Timeout

The remote procedure call mechanism used in Triton is not tolerant of long operations. Unfortunately, the "load_and_bind" operation is potentially a long operation, and so it can cause RPC timeout errors.

There are several work-arounds for this problem.

1. One solution is to check to see if the "load_and_bind" failed. If so, then "sleep" for a period of time (60 seconds is a good minimum) and then try to load it again. The second load should succeed. The sleep-load cycle can be repeated several times.

2. As mentioned above, triton will pre-load any object files specified as command line arguments. Thus, if one knows what object files need to be loaded, they can be specified on the command line. Then, later, when the attempt is made to "load_and_bind" that object file, it will generally succeed and not timeout because the file was already loaded. Note that the client still has to invoke the "load_and_bind" operation in order to bind the symbols into the catalog, it is just that the file will not need to be loaded since it is already there.

3. (Best) Set the TritonRPCTimeout property to a sufficiently large number.

13.2 Using Generic Classes

Programs dynamically loaded into Triton are free to define generic classes and classes derived from them. Their are some complications, though, in using methods in derived classes. Normal class methods take the instance as an implicit first argument. It turns out that methods

associated with generic classes take a special argument as a second implicit argument. This special argument contains information necessary to specialize the generic method to act as a method for some class derived from that generic class. The consequence for Triton is that it must be informed of the methods of derived classes that come from the generic class. To this end, the interface operation "create_method_derived" should be used to define every derived method that will be used by a derived class. In section 17, on using APPL/A, you can see an application of this where class "RELATION" is generic, and class "simprel" is a derived class with derived methods.

Derived methods also introduce another complication. If some of the arguments, or a result of a generic method depends upon the type of some generic parameter, then some care must be taken in defining the code for the generic function because Triton has only imperfect modelling of this feature. In particular, Triton is not capable of correctly handling these generic parameters. As a result, it is generally best to use the RLI_Q type for arguments and results of such methods.

As mentioned in section 12, you may also need to apply the "makeglobal" utility to an object file to be loaded. You must do this if the object file contains a derived class definition.

13.3 Using Subclasses

Similar to derived classes, inherited methods for subclasses must be explicitly defined as part of the subclass in the schema using "create_inherited_class." It is possible that in the future, this requirement can be lifted and automatic inheritance provided.

13.4 Destroying Schema Objects

You should be warned that this is operation has known faults, and so it can sometimes crash the server; beware.

14 Using Triton to Support APPL/A

One of the purposes of Triton is to provide support for APPL/A relations, both on the E side and on the Ada side. It does this by providing a generic E dbclass "APPLA_RELATION," and a corresponding generic Ada package "Triton_Relation." Each generic understands the protocol used by the other and so by appropriately instantiating both, the E side code can be used to provide persistent storage and access to APPL/A relations constructed using the Ada side generic package. This support is presently limited in that Triton relations do not support constraints or derived data; this is assumed to be provided by the APPL/A program. It is possible to use Triton triggers on relation methods and these may be able to replace or augment some APPL/A triggers.

14.1 Dbclass APPLA_RELATION

The file $TRITONHOME/include/appla_relation.h shows the class specification for the generic class "APPLA_RELATION." This generic class is parameterized by a class "TUPLE" defining the tuple structure. The file $TRITONHOME/include/appla_relation.h also defines a base class "APPLA_TUPLE" of which all tuple classes should be defined as a subtype. "APPLA_TUPLE" provides defaults for all the methods, required and optional, needed by "APPLA_RELATION."

In addition to the normal constructor and destructor methods, the APPLA_ RELATION class expects the tuple type to provide several methods for its use.

Fieldcount: This returns a constant which indicates how many fields are in the tuple. This is needed to decode the boolean arrays used in selection_test and selection_update (seee below).

Equal: The equal method takes a pointer to a tuple and compares it to "this" tuple. It returns TRUE (1) or FALSE (0) as the tuples are considered equal. Typically, this function compares each field of the tuples using the appropriate equality for the field type.

Selection_test: This method takes a tuple and a selection list, a vector of booleans (really integers) of length fieldcount(). The elements of the selection list that contain the value TRUE (1), determine which fields of the tuple argument and the "this" tuple are compared for equality. A value of TRUE or FALSE is returned.

Selection_update: This method takes a tuple and a selection list, and a vector of booleans. The elements of the selection list that contain the value TRUE (1), determine which fields of the "this" tuple are updated from corresponding fields of the tuple argument. No value is returned.

Edr: This function is used to linearize or de-linearize a tuple with respect to a QDR_Handle. When linearizing, the "this" tuple's fields are linearized into the QDR_Handle. When de-linearizing, the fields are extracted from the QDR_Handle. This function is called "edr" rather than, say, "qdr" to indicate that a possibly persistent object is being manipulated. See Section 15 on handling E pointers for a more detailed discussion.

A number of other methods of "TUPLE" are optional.

Print: This function prints out the value of a tuple on the file (type FILE*) passed as the argument to this function.

Write_tuple: This function writes some encoding of its tuple value onto the file (type FILE*) passed as the argument to this function. This choice of encoding is purely up to the writer of this tuple class. If one chooses to use a printable encoding, then it may be possible to share code with the print function. The intent of this function is to provide a method for saving the contents of a relation independent of Triton.

Read_tuple: This function is essentially the inverse of "read_tuple." It reads some encoding from the file (type FILE*) passed as the argument to this function and stores the values into its fields.

The file $TRITONHOME/src/appla_relation.e contains the code for the methods for the class APPLA_RELATION. The argument types should be noted in this file. All the functions that take arguments that depend in the tuple type are defined to be ¡encoded¿ (i.e., RLI_ENCODED). Also, the find function returns a result that depends on the tuple type, so it has ¡encoded¿ as its result type. This is one of the problems in using generic methods in Triton. Referring to the header file ($TRITONHOME/include/appla_relation.h) the definition of each method has its real argument type in a comment. Thus, method "insert" really is expecting a linearized tuple instance.

The "APPLA_RELATION" class provides a number of methods. A number of the methods correspond closely to the APPL/A definitions.

```
int insert(QDR_Handle);
int remove(QDR_Handle);
int update(QDR_Handle);
QDR_Handle find(QDR_Handle,QDR_Handle);
```

A number of additional methods are provides that may be useful for debugging.

Set_debug: This takes an integer, level, as its argument and sets the level of debug output for the relation instance to that level. It returns the old level as its result.

Clear: This clears the relation instance by deleting all the tuples in it.

Print: This takes the name of a file as its argument and places a printable representation of the contents of the relation into that file. If the string is null, then the default is standard output.

Write_relation: This takes the name of a file as its argument and writes the tuples in the relation to that file. using the "write_tuple" method of the tuple class.

Read_relation: This takes the name of a file as its argument, clears the relation, and retrieves tuples from the file using the "read_tuple" method of the tuple class.

Finally, a constructor method and a destructor method are part of the definition of "APPLA_RELATION."

The file $TRITONHOME/src/tests/two.e shows an example using the APPLA_RELATION generic to construct a simple relation. The tuple type is defined by the class "two_tuple" which has two integer fields: "key" and "field." The required methods are defined as part of this tuple class.

The derived class "two_relation" is declared as an instantiation of the generic class "APPLA_RELATION" parameterized by the class "two_tuple." Next, a persistent instance, "two," of this derived relation is declared. Finally, a function, "get_two," is defined to return a pointer to the persistent instance variable. This instance is needed as an argument to "evaluate" when invoking any of the methods of APPLA_RELATION. But Triton only allows values to be obtained as the result of a function evaluation, so a function must be provided to obtain this instance pointer.

The function "create_appla_relation_schema" in the file $TRITONHOME/src/tritonutil.e shows how the schema for the derived relation can be defined. It defines a class for the tuple type and then it calls "create_class_derived" to define the derived relation class. This is followed by calls to "create_method_derived" to define the relation methods, and calls to "create_formal" to define the formal argument types. Finally, a retrieve function is defined. This procedure assumes that the schema already contains the appropriate classes and methods for class APPLA_RELATION.

As mentioned in Section 12, you will need to apply the "makeglobal" utility to two.o so that it can be loaded correctly. In this case, the command would be:

```
makeglobal   two.o   _two_relation__gtbl   __two_relation__ctor
```

and after this, the file is ready for dynamic loading.

14.2 Package Triton_Relation

On the APPL/A (Ada) side, the assumption is that you have an APPL/A relation specification and you want to construct a body that will maintain the relation contents persistently in a Triton server. the basic mechanism for doing this is to define the entries into the relation body to invoke appropriate procedures in the generic package "Triton_Relation." This generic is contained in the files $TRITONHOME/src/ada/triton_relation.a and $TRITONHOME/src/ada/triton_relation.body.a.

This generic is parameterized by several arguments, not all of which are strictly required. For upward compatibility, you should not specify these parameters positionally, but rather by name (e.g., *Relation_Name =¿ "xxx"*).

Hostname: This specifies the hostname on which the server is running. With Triton version 0.7.2 and later, this parameter is no longer necessary and defaults to the null string. Instead, the server name is taken from the TritonHost property.

Relation_Name: This is the name of the APPL/A relation.

E_Relation_Name: This is the name of the APPL/A relation. This name is used as an argument to "create_appla_relation_schema" to create the Triton side schema for this relation. If this field has the value "X" then the corresponding E code should have the following definitions.

```
dbclass X_relation: APPLA_RELATION[...];
persistent X_relation X;
X_relation* get_X() { return &X;}
```

Fieldcount: This is an integer defining the number of fields in the tuple type for this relation. It should correspond to the value of the fieldcount method in the E code.

Loadfile: This is the name of the object file that results from compiling the E code for this relation. In versions prior to Triton 0.7.2, this should be an absolute path name so that it is independent of the the directory in which the server is executing. In Triton 0.7.2 and later, the TritonSearchPath property may be modified to contain the path and then only a relative name need be used here.

Deffile: This is the name of the definitions file that results from compiling the E code for this relation with the "+t" option. Typically this has the same value as the "loadfile" argument but with ".d" substituted for the ".o" ending.

Tupletype: This is the name of the APPL/A tuple type specified in the relation specification. It is, of course, translated into an Ada record type of the same name. It is assumed that the structure of this record corresponds to the tuple class defined in the E code. This correspondence is critical for the proper operation of this package.

Qdr_tupletype: This is a Q encode/decode procedure that is capable of linearizing and delinearizing an instance of the tuple type for remote procedure call transmission. It is assumed that this procedure uses and encoding that is compatible with the "edr()" entry of the tuple class in the E code.

NullTuple: This is an instance of TupleType with null values in all fields.

The files $TRITONHOME/src/tests/two.ap and $TRITONHOME/src/tests/two.body.ap shows the typical construction of an APPL/A relation body to use the generic package. This example shows the definition of a "two" relation that corresponds to the "two.e" code described in the previous section.

15 Handling E Pointers

If the client is written in any language but E, then handling E pointers is no particular problem; just treat them as you would any 16 byte structure. If, however, you are writing a client in E, then you must sometimes be careful in handling E pointers. In particular, you must remember that any time you take the address of a *db* type, including *dbvoid** (which is what a E pointer is), then the result is an E pointer, not a normal C pointer. The obvious place where this is an issue is in arguments to Q encode/decode functions. Consider the following E code procedure

```
int qdr_epointer(QDR_Handle qdrs, dbvoid** ptr)
{
    return qdr_opaque(qdrs,ptr,sizeof(dbvoid*));
}
```

This code will not work correctly because the qdr_opaque function expects a normal C pointer as its second argument. But the type "dbvoid**" is really an E pointer, and so it is not the proper argument to qdr_opaque.

The proper way to write this function is as follows.

```
struct EPOINTER { dbvoid* ep; };

int qdr_epointer(QDR_Handle qdrs, struct EPOINTER* ptr)
{
    return qdr_opaque(qdrs,ptr,sizeof(struct EPOINTER));
}
```

The reason that this works is that taking the address of a normal C type, a struct in this case, returns a normal C pointer. This is true even when the contents of the structure is an E pointer. If you examine the code of $TRITONHOME/src/tests/c1.e, you will see this fiddling with E pointers.

As an aside, the type "struct EPOINTER" is defined in the Triton interface header "Triton-Client.h." A function called "edr_epointer" is also defined in that interface and is the equivalent of the function defined above. Other "edr" functions are also defined: edr_integer, edr_floating, edr_pointer, edr_string, edr_bytes, and edr_array.

16 Utility Programs

Triton clients sometimes require use of some utility programs to accomplish some activities.

16.1 Makeglobal

This utility modifies the symbol table for the object file to make specified symbols externally accessible. As a rule, you should perform the following command for every derived class in the object file that will be defined in the schema:

```
makeglobal   <object_file_name>   _X__gtbl   __X__ctor
```

where "X" is the name of the derived class.

16.2 Tcmd

This utility provides a more-or-less convenient way to invoke many of the triton interface operations without having to write a specific program. The general form of its command line is one of the following:

```
tcmd   [<server>]   <cmd>   <arg1>...
tcmd   [<server>]
```

The server is the name of the host on which presumably is running the Triton server of interest. If no server is specified, then the value of the property "TritonHost" is used. If no command is given, then tcmd enters interactive mode and will accept commands, one per line of input, until the "quit" command is given.

Arguments are one of the following types:

keyword: There are a number of words and symbols reserved as keywords: help, quit, ping, create, destroy, load, unload, shutdown, immutable, print, report, debug, properties, property, type, class, generic, derived, function, method, inherited, formal, relation, relations, trigger, triggers, is, on, over, via, off, returns, return, all, set, ::, :, , -

Identifier: An identifier is any sequence of characters from upper or lower alphabetic or digits or one of the special characters with the constraint that it may not start with a digit. The set of special characters is: *!#$*& ?_/.+-*

Integer: Any sequence of digits, possibly with a leading plus or minus sign.

String: Any sequence of characters enclosed in double quotes. A Double quote may be included if it is preceded by the backslash as an escape character.

Literal: Any sequence of characters enclosed in single quotes. A single quote may be included if it is preceded by the backslash as an escape character.

There are several commands:

Help: Provide a detailed listing of the command for "**tcmd**."

Ping: Test to see if the server is responding. As a side effect, it will obtain the version of the server and print it out.

Load: Load a specified object file and optionally an associated definitions file into Triton. This command may suffer from the timeout problem.

Unload: Given the name of an object file, it unloads the registry stack through that file.

Shutdown: Shutdown the specified server in an orderly fashion.

Print: Cause the server to dump its schema onto its standard output. Note that this will in general not be your terminal.

Report: Cause the server to report to standard output, any unbound methods, functions, class sizes, and more. Note that this will in general not be to your terminal.

Create: Create a specified schema element. The options and syntax here are a bit tricky, so read the help carefully. All schema elements are specified by name, so all create commands require the specification of a name.

Destroy: Destroy a schema element.

Property: Tcmd allows query and modification of server properties using three command forms:

```
properties
property <property name>
property <property name> = <newvalue>
```

The first command prints the names and values of all properties known in the server. The second command prints the value of a specified property. The third command sets the value of a specified property and prints the old and new values. Both property names and values may be specified as identifiers, strings, literals, or integers. Thus the following commands are equivalent:

```
tcmd property TritonDebugLevel = "5"
tcmd property TritonDebugLevel = 5
```

17 Using Triton with Ada

There is an Ada version of the client interface to Triton that matches closely the C interface. This version has almost all of the same operations as are available through the C interface. Of course, the argument types are defined slightly differently, due to language differences. To see the interface, examine the file $TRITONHOME/src/ada/triton.a and $TRITONHOME/src/ada/tritonutil.a. In $TRITONHOME/src/tests, there a couple of test programs that illustrate the construction of Ada clients. A comparison with the corresponding E programs is instructive.

Beginning with version 0.7.2, all clients must be loaded with the client side library ("$TRITONHOME/lib/tritonlib_ada.a") in order to access the property facilities.

18 Changes from version 0.7.2

Version 0.7.3of Triton differs from version 0.7.2in the following ways.

1. The generic relation package for APPL/A relations has another parameter, *E_Relation_Name*, that stands for the name of the E-code relation. the parameter *Relation_Name* now stands for the name of the APPL/A relation. For backward compatability, E_Relation_Name uses Relation_Name as its default value. The primary reason for thsi change is to provide better client side debug output.

2. The generic relation package for APPL/A relations has another parameter: *NullTuple*. This is an instance of TupleType with null values in all fields. It is used inside the generic body to make sure that the initial value of the output result tuple in find has a null value.

3. The APPLADebugLevel property was added to the client side property list.

4. The set of allowable special characters in identifiers in tcmd has been increased.

5. A literal argument type (in single quotes) has been added to tcmd.

Property Name	Global Variable	Obsolete	Mutable	Type
ClientVersion	ClientVersion	No	No	String
TritonRPCProtocol	ServerRPCProtocol	No	Yes	String
TritonRPCNumber	ServerRPCNumber	No	Yes	Integer
TritonRPCVersion	ServerRPCVersion	No	Yes	Integer
TritonHost	ServerHost	No	Yes	String
TritonRPCTimeout	ServerRPCTimeout	No	Yes	Integer
ClientAPPLADebugLevel	ClientAPPLADebugLevel	No	Yes	Integer

Property Name	Environment Variables	Default
ClientVersion	N.A.	0.7.3
TritonRPCProtocol	TritonRPCProtocol, TRITONRPCPROTOCOL	"TCP"
TritonRPCNumber	TritonRPCNumber, TRITONRPCNUMBER	2
TritonRPCVersion	TritonRPCVersion, TRITONRPCVERSION	1
TritonHost	TritonHost, TRITONHOST	Same machine as client
TritonRPCTimeout	TritonRPCTIMEOUT, TRITONRPCTIMEOUT	100
ClientAPPLADebugLevel	ClientAPPLADebugLevel	0

Table 1: Client Properties.

Property Name	Global Variable	Obsolete	Mutable	Type
TritonVersion	TritonVersion	No	No	String
EPath	EPath	No	Yes	String
TritonPath	TritonPath	No	Yes	String
TritonDebugLevel	TritonDebugLevel	No	Yes	Integer
APPLADebugLevel	APPLADebugLevel	No	Yes	Integer
TritonRPCProtocol	TritonRPCProtocol	No	No	String
TritonRPCNumber	TritonRPCNumber	No	No	Integer
TritonRPCVersion	TritonRPCVersion	No	No	Integer
TritonHost	TritonHost	No	No	String
TritonSearchPath	TritonSearchPath	No	Yes	Pathlist

Property Name	Environment Variables	Default
TritonVersion	N.A.	0.7.3
EPath	EPATH, Epath	Makefile constant
TritonPath	TritonPath,TRITONPATH	Makefile constant
TritonDebugLevel	TritonDebugLevel	0
APPLADebugLevel	APPLADebugLevel	0
TritonRPCProtocol	TritonRPCProtocol, TRITONRPCPROTOCOL	"UDP"
TritonRPCNumber	TritonRPCNumber, TRITONRPCNUMBER	2
TritonRPCVersion	TritonRPCVersion, TRITONRPCVERSION	1
TritonHost	TritonHost, TRITONHOST	Server machine
TritonSearchPath	TritonSearchPath, TRITONSEARCHPATH	Makefile constant

Table 2: Server Properties.

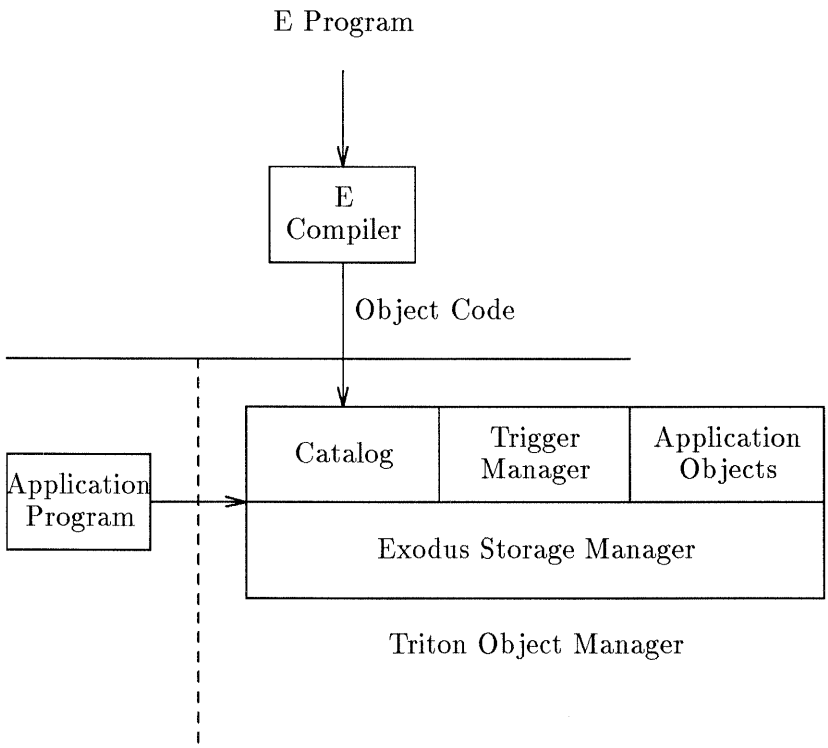

Figure 1: Triton Architecture.

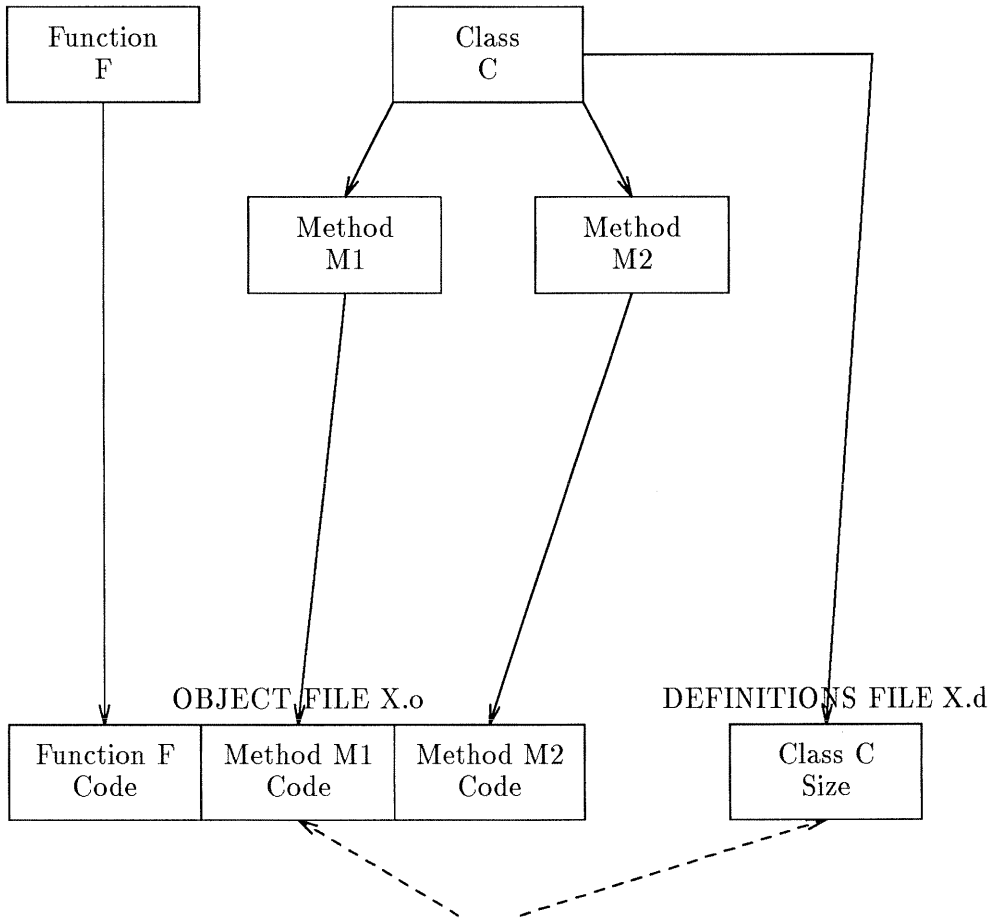

Figure 2: Schema and Object File Correspondence

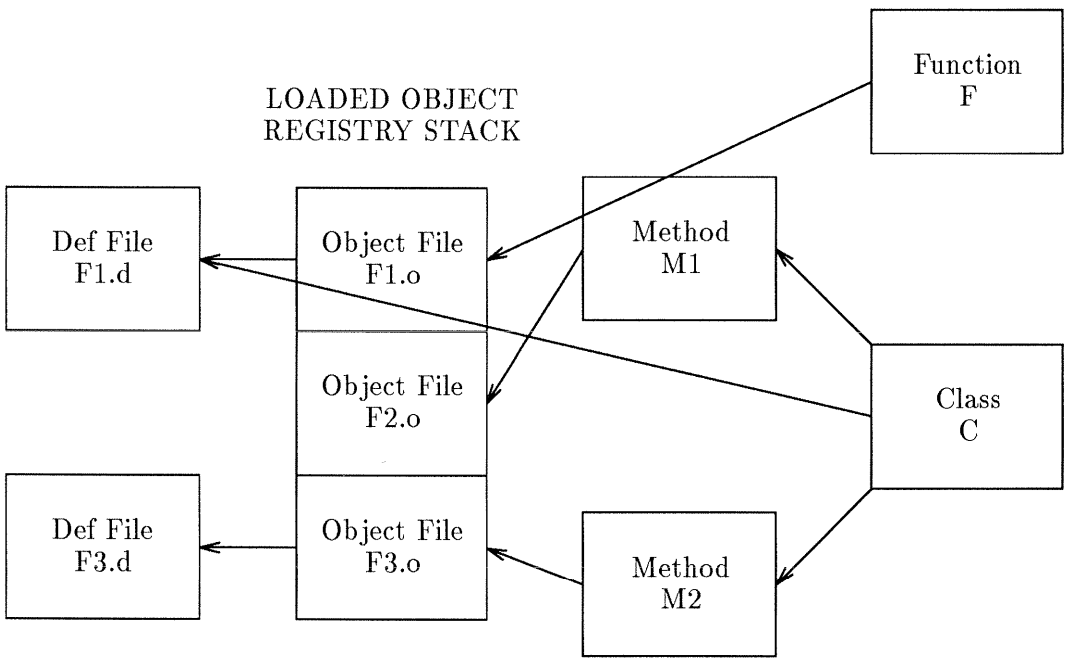

Figure 3: Loaded Files Stack