OSF DCE SIG **N. Mishkin (HP)**
Request For Comments: 21.0 **November 1992**

DCE RPC API EXTENSIONS FOR MODULAR SERVERS

1. INTRODUCTION: PROBLEM AND MOTIVATION

The DCE RPC API does not support modularity when it comes to servers: a single process can be at most one server. This deficiency was known at the time the API was designed. We chose to defer correcting it — even though we knew the general outline of the solution — since it seemed to be a relatively low priority problem. People generally understand why it must be possible to write local libraries that transparently are RPC clients; seeing that local libraries also occasionally want to be RPC servers (and hence why the server-related part of the API must support modularity) is somewhat harder. In the time since the API was designed, we've seen at least a couple of cases where support for such ''modular servers'' (also called ''packaged servers'') would be very useful.

Consider the following example that requires a modular API. Suppose we want to build a local library that has the following function in it:

```
void PrintFile
(
    char *fileName,
    void (*notificationFunction)(int printerCompletionStatus)
);
```

This function takes the name of a file to print and a pointer to a function that should be called when the file has been printed. (`PrintFile()` returns as soon as the file is queued to the printer and does not wait for the file to actually be printed.) Assume that the printer is remote and thus that a call to `PrintFile()` turns into a remote call to some print server.

Here's a sketch of how we might implement `PrintFile()`:

(a) Spawn a thread that becomes an RPC server for a ''notification IDL interface'' — i.e., calls `rpc_server_use_protseq()`, `rpc_server_register_if()`, `rpc_server_inq_bindings()`, and `rpc_server_listen()`.

(b) When we make the remote call to the print server, pass the bindings (presumably converted to string bindings or towers) obtained above as a parameter to the remote call.

(c) When the print server completes the print job, it makes a remote call back to the initiator of the print request using the notification IDL interface and at least one of the above bindings.

(d) When the initiator of the request receives the remote call, it calls through the `notificationFunction()` pointer that was passed to `PrintFile()`.

Notice that the caller of `PrintFile()` has no idea that any RPC work — either client or server — is going on on behalf of its call. The client just makes a local call and later its notification function gets called.

The problem with this implementation is that if the user of `PrintFile()` also happens to want to be an RPC server in its own right, then the client's use of the `rpc_server_...()` functions will conflict with `PrintFile()`'s use of those functions.

2. SOLUTION

The way to fix the lack of modularity support in the DCE RPC API is augment the API with a new set of calls that parallel the current set of `rpc_server_...()` functions. We call these the `rpc_mserver_...()` functions. One can think of the relationship between these two sets of functions as being the same as the relationship between the C runtime library standard I/O functions that *don't* take a `FILE *` (e.g., `printf()`) and the functions that *do* (e.g., `fprintf()`).

The API extensions follow in several sections.

2.1. Types, Constructors, and Destructors

The following functions and types create and destroy ‘‘server objects’’ and are not parallel with any existing functions in the API:

```
/*
 * A handle onto a server object.
 */
typedef rpc_opaque_ptr_t    rpc_mserver_handle_t,
                            *rpc_mserver_handle_p_t;

/*
 * A server handle that refers to the distinguished server object
 * (i.e., the one manipulated by the rpc_server_...() functions).
 */
const long  rpc_mserver_distinguished_server    = 0;

/*
 * R P C _ M S E R V E R _ A L L O C
 *
 * Create a new server object and return a handle to it.
 */
void rpc_mserver_alloc
(
    [out]       rpc_mserver_handle_t    *server_handle,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ F R E E
 *
 * Free a previously created server object.
 */
void rpc_mserver_create
(
    [out]       rpc_mserver_handle_t    *server_handle,
    [out]       unsigned32              *status
);
```

2.2. Parallel Extensions

The following functions are parallel to the existing `rpc_server_...()` functions:

```
/*
 * R P C _ M S E R V E R _ I N Q _ B I N D I N G S
 *
 * Return the bindings in the local process to which RPCs may be
 * made.  Note that object UUIDs are not part of these bindings.
 */
void rpc_mserver_inq_bindings
(
    [in]        rpc_mserver_handle_t    server_handle,
    [out]       rpc_binding_vector_p_t  *binding_vector,
    [out]       unsigned32              *status
);
```

```
/*
 * R P C _ M S E R V E R _ L I S T E N
 *
 * This routine tells the RPC runtime to being listening for RPCs (in
 * any of the registered interfaces) on all protocol sequences
 * previously registered with calls to rpc_protseq_register() or
 * rpc_protseq_register_wk().  The maximum number of concurrent calls
 * the RPC runtime can handle is given.  Note that this routine will
 * not return until either a fault occurs or a shutdown is requested.
 */
void rpc_mserver_listen
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned32              max_calls_exec,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ U S E _ A L L _ P R O T S E Q S
 *
 * This routine tells the RPC runtime to listen for RPCs on all
 * supported (by both the RPC runtime and the operating system)
 * protocol sequences.
 */
void rpc_mserver_use_all_protseqs
(
    [in]        unsigned32              max_call_requests,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ U S E _ P R O T S E Q
 *
 * This routine tells the RPC runtime to listen for RPCs on the
 * protocol sequence given.  Note that this protocol sequence must be
 * supported by both the RPC runtime and the operating system.
 */
void rpc_mserver_use_protseq
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned_char_p_t       protseq,
    [in]        unsigned32              max_call_requests,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ U S E _ P R O T S E Q _ E P
 *
 * This routine tells the RPC runtime to listen for RPCs on the
 * protocol sequence given.  Part of the address (endpoint) to be
 * listened on is provided by the caller.  Note that this protocol
 * sequence must be supported by both the RPC runtime and the
 * operating system.
 */
void rpc_mserver_use_protseq_ep
```

```
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned_char_p_t       protseq,
    [in]        unsigned32              max_call_requests,
    [in]        unsigned_char_p_t       endpoint,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ U S E _ P R O T S E Q _ I F
 *
 * This routine tells the RPC runtime to listen for RPCs on the
 * protocol sequence given.  Part of the address (endpoint) to be
 * listened on will be extracted from the interface spec handle.
 * Note that this protocol sequence must be supported by both the RPC
 * runtime and the operating system.
 */
void rpc_mserver_use_protseq_if
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned_char_p_t       protseq,
    [in]        unsigned32              max_call_requests,
    [in]        rpc_if_handle_t         if_spec,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ U S E _ A L L _ P R O T S E Q S _ I F
 *
 * This routine tells the RPC runtime to listen for RPCs on all the
 * protocol sequences for which the specified interface has
 * well-known endpoints.
 */
void rpc_mserver_use_all_protseqs_if
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned32              max_call_requests,
    [in]        rpc_if_handle_t         if_spec,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ R E G I S T E R _ I F
 *
 * Register an interface with the RPC runtime.  This includes the
 * interface specification handle which contains the interface UUID
 * and version, the server stub Entry Point Vector for this
 * interface, a manager Entry Point Vector for this interface and the
 * type UUID associated with this manager EPV.  This routine may be
 * called multiple times with the same interface spec handle but
 * different manager EPVs and type UUIDs.  The RPC runtime will group
 * these manager EPVs and type UUIDs with the single interface spec
 * handle and server stub EPV.  See rpc_object_register() for an
 * explanation of the purpose of providing a type UUID.
 */
```

```
void rpc_mserver_register_if
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        rpc_if_handle_t         if_spec,
    [in]        uuid_p_t                mgr_type_uuid,
    [in]        rpc_mgr_epv_t           mgr_epv,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ U N R E G I S T E R _ I F
 *
 * Unregister from the RPC runtime an interface spec handle, its
 * server stub Entry Point Vector and all manager Entry Point Vectors
 * for this interface.
 */
void rpc_mserver_unregister_if
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        rpc_if_handle_t         if_spec,
    [in]        uuid_p_t                mgr_type_uuid,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ I N Q _ I F
 *
 * Given an interface spec and type ID, return the manager EPV that
 * has been registered for them (if any).
 */
void rpc_mserver_inq_if
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        rpc_if_handle_t         if_spec,
    [in]        uuid_p_t                mgr_type_uuid,
    [out]       rpc_mgr_epv_t           *mgr_epv,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ R E G I S T E R _ A U T H _ I N F O
 *
 * Register authentication information with the RPC runtime.
 */
void rpc_mserver_register_auth_info
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned_char_p_t       server_princ_name,
    [in]        unsigned32              auth_svc,
    [in]        rpc_auth_key_retrieval_fn_t
                                        get_key_func,
    [in]        void                    *arg,
    [out]       unsigned32              *st
);
```

2.3. Management Extensions

The following are functions related to management. They have analogs in the current API (and the names of those functions should probably have included the string ``server'', subject to symbol name length limitations[1]):

```
/*
 * R P C _ M G M T _ M S E R V E R _ I N Q _ I F _ I D S
 *
 * Obtain a vector of interface identifications listing the
 * interfaces registered with the RPC runtime.  If a server has not
 * registered any interfaces this routine will return an
 * rpc_s_no_interfaces status code and a NULL if_id_vector.  The
 * application is responsible for calling rpc_if_id_vector_free() to
 * release the memory used by the vector.
 *
 * Unlike rpc_mgmt_inq_if_ids(), this function NEVER refers to a
 * remote server.
 */
void rpc_mgmt_mserver_inq_if_ids
(
    [in]        rpc_mserver_handle_t    server_handle,
    [out]       rpc_if_id_vector_p_t    *if_id_vector,
    [out]       unsigned32              *status
);

/*
 * R P C _ M G M T _ M S E R V E R _ I N Q _ S T A T S
 *
 * Obtain statistics about the specified server from the RPC runtime.
 * Each element in the returned argument contains an integer value
 * which can be indexed using the defined statistics constants.
 *
 * Unlike rpc_mgmt_inq_stats(), this function NEVER refers to a
 * remote server.
 */
void rpc_mgmt_mserver_inq_stats
(
    [in]        rpc_mserver_handle_t    server_handle,
    [out]       rpc_stats_vector_p_t    *statistics,
    [out]       unsigned32              *status
);

/*
 * R P C _ M G M T _ M S E R V E R _ I S _ L I S T E N
 *
 * Determine if the specified server is listening for remote
 * procedure calls.
 *
```

1. The DCE RPC API has a regularity in its function names. The basic scheme is: `<prefix>_<object>_<operation>`. Valid values for `<prefix>` are: `rpc`, `rpc_mgmt`, `rpc_ns`. Valid values for `<object>` are: `binding`, `if`, `if_id`, `network` (sort of), `protseq_vector`, `server`, `mserver` (now), `ns_entry` (`rpc_ns` prefix only), `group` (`rpc_ns` prefix only), `profile` (`rpc_ns` prefix only). The API is not completely consistent, especially in the area of the managment functions.

```
 * Unlike rpc_mgmt_is_server_listening(), this function NEVER refers
 * to a remote server.
 */
boolean32 rpc_mgmt_mserver_is_listen
(
    [in]        rpc_mserver_handle_t    server_handle,
    [out]       unsigned32              *status
);

/*
 * R P C _ M G M T _ M S E R V E R _ S E T _ S T A C K _ S I Z E
 *
 * Set the value that the RPC runtime is to use in specifying the
 * thread stack size when creating call threads.  This value will be
 * applied to all threads created for the server.
 */
void rpc_mgmt_mserver_set_stack_size
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned32              thread_stack_size,
    [out]       unsigned32              *status
);

/*
 * R P C _ M G M T _ M S E R V E R _ S T O P _ L I S T E N
 *
 * Direct a server to stop listening for remote procedure calls.  On
 * receipt of a stop listening request the RPC runtime stops
 * accepting new remote procedure calls for all registered
 * interfaces.  Executing calls are allowed to complete, including
 * callbacks.  After all executing calls complete the
 * rpc_server_listen() routine returns to the caller.
 *
 * Unlike rpc_mgmt_stop_server_listening(), this function NEVER
 * refers to a remote server.
 */
void rpc_mgmt_mserver_stop_listen
(
    [in]        rpc_mserver_handle_t    server_handle,
    [out]       unsigned32              *status
);

/*
 * R P C _ M G M T _ M S E R V E R _ S E T _ A U T H Z _ F N
 *
 * Specify the application function that the RPC runtime should call
 * when it receives remote request to do a management operation.  The
 * application function can decide whether the operation should be
 * allowed to proceed.
 */
void rpc_mgmt_mserver_set_authz_fn
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        rpc_mgmt_authorization_fn_t
                                        authorization_fn_arg,
```

```
    [out]       unsigned32              *status
);

/*
 * R P C _ M G M T _ M S E R V E R _ I N Q _ P R I N C _ N A M E
 *
 * Obtain server principal name information for the specified server
 * from the RPC runtime.
 *
 * Unlike rpc_mgmt_inq_server_princ_name(), this function NEVER
 * refers to a remote server.
 */
void rpc_mgmt_mserver_inq_princ_name
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        unsigned32              authn_svc,
    [out]       unsigned_char_p_t       *server_princ_name,
    [out]       unsigned32              *status
);
```

2.4. Object Extensions

The following are functions related to objects and types. They have analogs in the current API (and the names of those functions should probably have included the string ‘‘`server`’’, subject to symbol name length limitations):

```
/*
 * R P C _ M S E R V E R _ O B J E C T _ S E T _ T Y P E
 *
 * Set the type UUID associated with an object UUID in the RPC
 * runtime.  This routine, used in conjunction with
 * rpc_if_register(), allows a server to support multiple
 * implementations of the same interface.  The RPC runtime will
 * dispatch to a specific implementation, contained in a manager
 * Entry Point Vector, based on the object UUID contained in the
 * binding of the RPC.  The RPC runtime, using the results of a call
 * to this routine, will determine the type UUID of the object UUID.
 * A specific manager Entry Point Vector of this type UUID can then
 * be found using the results of a call to the rpc_if_register()
 * routine.
 */
void rpc_mserver_object_set_type
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        uuid_p_t                object_uuid,
    [in]        uuid_p_t                type_uuid,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ O B J E C T _ I N Q _ T Y P E
 *
 * Given an object ID, return its type ID.
 */
void rpc_mserver_object_inq_type
(
    [in]        rpc_mserver_handle_t    server_handle,
```

```
    [in]        uuid_p_t                object_uuid,
    [out]       uuid_t                  *type_uuid,
    [out]       unsigned32              *status
);

/*
 * R P C _ M S E R V E R _ O B J E C T _ S E T _ I N Q _ F N
 *
 * Supply a function that is called by the runtime to determine the
 * type of objects that have not been set by
 * rpc_mserver_object_set_type().
 */
void rpc_mserver_object_set_inq_fn
(
    [in]        rpc_mserver_handle_t    server_handle,
    [in]        rpc_object_inq_fn_t     inq_fn,
    [out]       unsigned32              *status
);
```

3. IMPLEMENTATION

Implementing the API extensions is not a small amount of work, however it should be relatively straightforward. One way of approaching the task is to think of moving some large number of the RPC runtime's top level static or global (i.e., non-automatic and non-heap) storage to be fields in a heap-allocated server object struct. One might not want to approach this blindly though. For example, it might be appropriate to retain a single table containing all the file descriptors — from *all* the server objects — that the listener thread should `select()` on. Entries in this table would be tagged by which server object they came from so they could be removed selectively.

AUTHOR'S ADDRESS

Nathaniel Mishkin
Distributed Object Computing Program
Hewlett-Packard Co.
250 Apollo Drive
Chelmsford, MA 01824
USA

Internet email: mishkin@apollo.hp.com
Telephone: +1-508-436-4353