One DCE/RPC server to serve them all

Samuel Cabrero
scabrero@suse.com
SUSE

DCE/RPC

DCE / RPC

DCE – Distributed Computing Environment

- Framework and toolkit to develop client/server applications
- Developed by the OSF (Open Software Foundation) in early 1990s
- DCE/RPC (Remote Procedure Call) is part of the framework

RPC – Remote Procedure Calls

- Infrastructure to call a function on a remote server
- Remote is connected over different kind of transports
  - SMB Named pipes → ncacn_np
  - TCP/IP → ncacn_ip_tcp
  - Others ncacn_http, ...

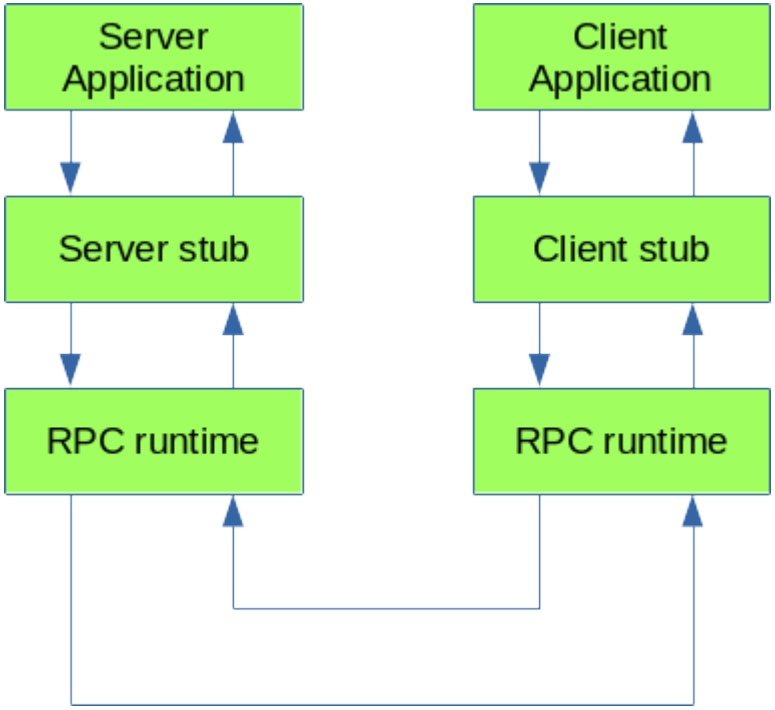
Microsoft extensions

- Documented in MS-RPCE

Remote Procedure Calls

A RPC call traverse five code blocks

- Client application
- Client stub
- RPC runtime
- Server stub
- Server application



Stubs are generated by compiling a description of the interface (IDL) with an IDL compiler

IDL

```
[
  uuid("60a15ec5-4de8-11d7-a637-005056a20182"),
  endpoint("ncacn_np:[\\pipe\\rpcecho]", "ncacn_ip_tcp:", "ncalrpc:"),
  pointer_default(unique),
  version(1.0),
  helpstring("Simple echo pipe")
]
interface rpcecho
{
        /* Add one to an integer */
        void echo_AddOne(
                [in] uint32 in_data,
                [out] uint32 *out_data
        );
}
```

Client stub

To make an RPC call, the client invokes a function in the client stub

The stub converts local application data into network data for transmission (marshalling)

Asks the RPC runtime to send the packets

```
NTSTATUS dcerpc_echo_AddOne(struct dcerpc_binding_handle *h,
                            TALLOC_CTX *mem_ctx,
                            uint32_t _in_data /* [in]  */,
                            uint32_t *_out_data /* [out] [ref] */);
```

Endpoints and interfaces

An endpoint can be a port or a pipe and provide several interfaces

- ncacn_np:[\\pipe\netlogon] serves netlogon interface and lsa

A interface is the RPC service provided in an endpoint

Endpoints can be dinamically allocated

- The Endpoint Mapper provide information about the endpoints

DCE/RPC in Samba

The journey

2003 – Samba 3.0.0

- Hand written marshalling code
- Only implemented what was required by windows clients

2003 – Samba 4.0 development starts

2006 – Samba 4.0.0 TP1 (Technology Preview 1)

- New DCE/RPC server infrastructure, asynchronous design, single process
- Marshalling code is autogenerated thanks to the new IDL compiler (PIDL)

Why a new server for S4

Newer services require asynchronous processing

- [MS-SWN] Service Witness Protocol
- [MS-PAR] Print System Asynchronous Remote Protocol
- [MS-FRS2] Distributed File system Replication Protocol

Support for association groups

Easier to maintain security

- Abstracted by gensec

Header signing

Verification trailer

Bindtime feature negotiation

The journey

2008 – Samba 3.2

- PIDL backported, hand written marshalling code for some services replaced by autogenerated code

2009 – Samba 3.4

- Remaining hand written marshalling code replaced by autogenerated code
- RPC services can be “moved” to external processes
- NPA (Named Pipe Auth) abstraction use Unix sockets to implement SMB named pipes

2011 – Samba 3.6

- EPM implemented
- PIDL generates one set of client stubs, common for S3 and S4, based on binding handles abstraction
- Binding handles have several implementations

2012 – Samba 4.0

Current status

Two servers implementations

- Samba 3 server
  - Runs different processing loops depending on the transport
    - named_pipe_packet_process
    - dcerpc_ncacn_packet_process
  - Synchronous
  - Services can run embedded or external (plus preforking)

- Samba 4 server
  - Asynchronous
  - Single process (except netlogon)

The aim is to merge the good parts of all implementations together and extend the result to be more feature complete.

How to do it

Proposal

Extract the RPC core from S4 server and move it to a library

Modify S3 initialization

Modify S3 connection handlers

Generate a new set of server stubs

S3 server – Initialization

Samba 3 – `main()`

- Fork EPMD if enabled

- `start_epmd()`
  - Fork a child
  - `rpc_epmapper_init()`
    - `rpc_srv_register`
  - Setup ncacn_ip_tcp socket
  - Setup ncalrpc socket
  - Setup ncacn_np socket

S3 server – Initialization

Samba 3 – `main()`

- Fork EPMD if enabled
- Initialize embedded services

- `dcesrv_ep_setup()`
  - `rpc_setup_`*`service`*`()`
    - If the service is embedded
      - `rpc_`*`service`*`_init()`
        - `rpc_srv_register()`
  - `Load and setup RPC modules`

S3 server – Initialization

Samba 3 – `main()`

- Fork EPMD if enabled
- Initialize endpoints
- Fork daemons for enabled external services
  - LSASD (Local Security Authority)
  - SPOOLSSD (Network printing spooler)
  - FSSD (File Server Remove VSS)
  - MDSSD (Spotlight, Metadata Search Service)

- `start_lsassd()`
- Fork a child
  - `rpc_lsarpc_init()`
    - `rpc_srv_register()`
  - `rpc_samr_init()`
    - `rpc_srv_register()`
  - `rpc_netlogon_init()`
    - `rpc_srv_register()`
  - Setup ncacn_ip_tcp sockets
  - Setup ncalrpc sockets
  - Setup ncacn_np sockets

S4 server – Initialization

Task initialization

1. Load RPC modules
2. Run module initialization functions – E.g. `dcerpc_server_rpcecho_init`
   1. Register the endpoint server – `dcerpc_register_ep_server`
3. Initialize server context – `dcesrv_init_context`
   1. Initialize all endpoint servers enabled in smb.conf
      1. Initialization function creates and registers the declared endpoints in the server context
      2. Registers the interface in the endpoints
4. Initialize endpoints – `dcesrv_init_endpoints`
   1. Setup the sockets for each endpoint registered in the server context
      - dcesrv_add_ep_unix
      - dcesrv_add_ep_ncalrpc
      - dcesrv_add_ep_tcp
      - dcesrv_add_ep_np

S3 server – Initialization proposal

1. Start EPMD if enabled

1. Fork
2. Register “epmapper” endpoint server
3. Initialize server context
4. Initialize “epmapper” endpoint server
   1. Create and register the declared endpoints in the server context
   2. Register the interface in the endpoints
5. Initialize endpoints
   1. Setup the sockets for each registered endpoint in the server context

2. Initialize embedded services

3. Start daemons for external services

S3 server – Initialization proposal

1. Start EPMD if enabled

2. Initialize embedded services

1. For each embedded service, register endpoint server
2. Load and setup RPC modules
3. Initialize server context
4. Initialize all registered endpoint servers
   1. Create and register the declared endpoints in the server context
   2. Register the interface in the endpoints
5. Initialize all registered endpoints in the server context
   1. Setup the sockets depending on the transport
   2. If transport is NCACN_NP, register in the endpoint mapper

3. Start daemons for external services

S3 server – Initialization proposal

1. Start EPMD if enabled

2. Initialize embedded services

3. Start daemons for external services

1. Fork
2. Register required endpoint servers
3. Initialize server context
4. Initialize required endpoint servers
   1. Create and register the declared endpoints in the server context
   2. Register the interface in the endpoints
5. Initialize endpoints
   1. Setup the sockets for each registered endpoint in the server context

Context initialization

The initialization function has a new argument, a pointer to a dcesrv_context_callbacks struct

This struct hold pointers to functions whose implementation differs in S3 and S4

```
struct dcesrv_context_callbacks {
        struct {
                void (*successful_authz)(struct dcesrv_call_state *);
        } log;
        struct {
                NTSTATUS (*gensec_prepare)(TALLOC_CTX *mem_ctx,
                                           struct dcesrv_call_state *call,
                                           struct gensec_security **out);
        } auth;
};
```

Connection handlers

Socket creation functions setup a listener function

- dcesrv_setup_ncalrpc_socket → dcesrv_ncalrpc_listener
- dcesrv_setup_ncacn_ip_tcp_socket → dcesrv_ncacn_ip_tcp_listener
- dcesrv_setup_ncacn_np_socket → dcesrv_ncacn_np_listener

The listener functions accept the connection on the socket

The accept handler initialize both, S3 and S4 structures

- S3 dcerpc_ncacn_conn
- S3 pipes_struct, stored in dcerpc_ncacn_conn
- S4 dcesrv_connection
- Store dcerpc_ncacn_conn in dcesrv_connection.transport.private_data

Start the processing loop

Processing loop

Move required parts of S4 server to a new “core” library

- Initialization code and connection handlers remain specific for each implementation
- Functions called by the loop whose implementation differ called through the dcesrv_context_callbacks struct

Provide a function to start the loop, dcesrv_connection_loop_start

- From this point on, the processing loop is common to both S3 and S4

Write a new PIDL module to generate code compatible with S3 service implementations (ServerCompat)

ServerCompat PIDL module

Based in the S4 server stub generator

Endpoint server initialization

- For each declared endpoint in IDL
  - Registers the interface in the server context endpoints
    - If the service is embedded, register only in ncacn_np transport endpoint

Bind

- Retrieve pipes_struct from dcesrv_connection
- Initialize pipes_struct handles and pipe_rpc_fns context

Dispatching

- Retrieve pipes_struct from dcesrv_connection
- Update pipes_struct fields with dcesrv_call_state info
- Become authenticated pipe user
- Call S3 service implementation
- Unbecome authenticated pipe user

Internal RPC dispatching (rpcint_binding_handle)

First approach

- Endpoint server initialization in server stub registers the api_struct too
- Craft a pipes_struct and dispatch through api_struct

Second approach:

- Add a local dispatch function to ServerCompat PIDL module
  - rpcint gets the endpoint server by NDR table name
  - rpcint gets the interface by NDR table syntax ID
  - rpcint crafts the pipes_struct
  - rpcint calls local dispatching iface.local(p, opnum, mem_ctx, in_data, out_data)
  - Local dispatching
    - Pull
    - Dispatch
    - Reply
    - push

Summary – The plan

Step 1

- Prepare S3 and S4 code base

Step 2

- Write the CompatServer PIDL module
- Drop S3 loop

Step 3

- Add local dispatching to CompatServer
- Drop api_struct and S3 server stubs

Step 4

- Share service implementations when possible (ex. epmapper, mgmt, rpcecho)
- Rewrite service implementations (pipes_struct, handles)
- Drop pipes_struct and S3 handles implementation

DONE

Preparation for S3 code

Objectives

- Unify ncacn_np and ncacn_ip_tcp processing loops
- Unify named_pipe_client and dcerpc_ncacn_conn

45 patches, 63 files changed, 1705 insertions(+), 2135 deletions(-)

- Rename socket creation functions and return NTSTATUS
  - dcesrv_[setup | create]_[transport]_socket
- Rename listener functions
  - dcesrv_[transport]_listener
- Unify termination and disconnection functions
- Fix strict aliasing issues with sockets API
- Remove struct named_pipe_client and use struct dcerpc_ncacn_conn
- Remove named_pipe_packet_process loop, use dcerpc_ncacn_packet process
- Prepare preforking process model
  - Associate private data to listening socket, will be used to store the associated endpoint
- Minor fixes

https://gitlab.com/samba-team/devel/samba/tree/scabrero-rpc-merge-s3-prep-v4

Preparation for S4 code

Objective

- Move server code to core library
- Hide S4 specific structs

26 patches, 55 files changed, 4279 insertions(+), 4591 deletions(-)

- Hide imessaging_context and “server_id” behind “getter” functions, get from transport private data (stream_connection)
- Create “dcesrv_context_callbacks” to hold function pointers fo those functions which will diverge between S3 and S4:
  - successful_authz logging function
  - gensec_prepare
- Move core functions to librpc/rpc
- Split dcesrv_context initialization and registered endpoint servers initialization
  - S3 needs to initialize all registered endpoint servers running in “embedded” mode

https://gitlab.com/samba-team/devel/samba/tree/scabrero-rpc-merge-s4-prep-v4

Steps 2, 3, 3.1

78 patches, 143 files changed, 10131 insertions(+), 11321 deletions(-)

- “CompatServer” PIDL
  - Generate server stub compatible with S3
- Refactor RPC server initialization code
  - Register endpoint servers
  - Initialize endpoint servers → Register the interface in each endpoint
  - Initialize endpoints → Create the sockets for each endpoint
  - Drop dcerpc_binding_vector, create the dcerpc_binding to register the endpoints in the endpoint mapper using the endpoint description from the interface.
  - Switch to core server loop
  - Remove S3 loop
  - Run raw_protocol tests against NT4_DC
- Step 3.1
  - Add local dispatching to CompatServer, drop api_struct and S3 generated stubs
  - Example: Share the RPCECHO implementation

https://gitlab.com/samba-team/devel/samba/tree/scabrero-rpc-merge-v4

Tests

samba.tests.dcerpc.raw_protocol passes in NT4_DC

samba4.rpc.echo.*on.*with.object.echo passes in NT4_DC

TODO

TODO

S3 “process model”

- Association groups require single process
- LSASD daemon run in preforking mode
  - Run netlogon in preforking mode
  - Run SAMR and LSARPC embedded

Drop pipes_struct

- Rewrite service implementations, can be done one by one

Share service implementations when possible

Questions

SUSE®

We adapt. You succeed.