An Introduction to Resource Plugin Development

iRODS Consortium Development Team

What Are iRODS Plugins? (Part 1)

Plugins are dynamically loaded code which implements interfaces through which one can influence runtime behavior. Types of plugins include:

Resources, Authentication, Network, and Microservices

The use of plugins:

- Provides a method of confining functionality to an easily testable dynamically loaded library
- Provides ability to change runtime behavior in iRODS without recompiling all of the servers

Plugins are derived C++ Objects which are created via a well known factory method loaded directly from a dynamic library

What Are iRODS Plugins? (Part 2)

- A vehicle for mapping new functionality from the dynamic library to understood *Operations* used by iRODS
- Plugins provide a place to maintain state across *Operations* during an iRODS transaction via a heterogeneous property map
- Plugins provide the functionality, or ‘How’ in the system
- Plugins provide opportunity to leverage type and inheritance with a C++ Object Model, or the ‘What’

First Class Objects: The ‘What’

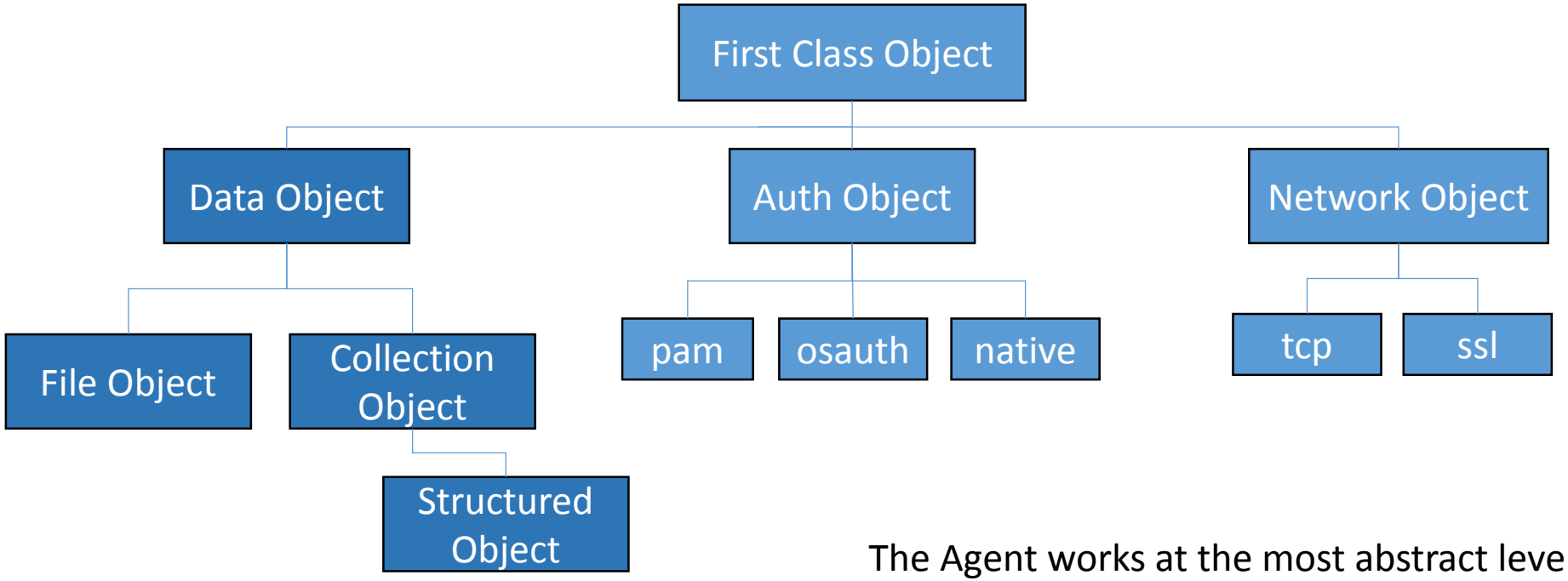
- Create a formalized type hierarchy for iRODS

  Informally exists today: e.g., Special Collections
  - Mounted collection
  - Linked collection
  - Structured File Object
- First Class Objects resolve plugins for a given interface:

  e.g., Resource, Network, Authentication, Microservices

  If First Class Objects are the ‘What’ and plugins are the ‘How’
  e.g., file_object::resolve(“resource interface”) → Resource Plugin

First Class Objects: The Object Model

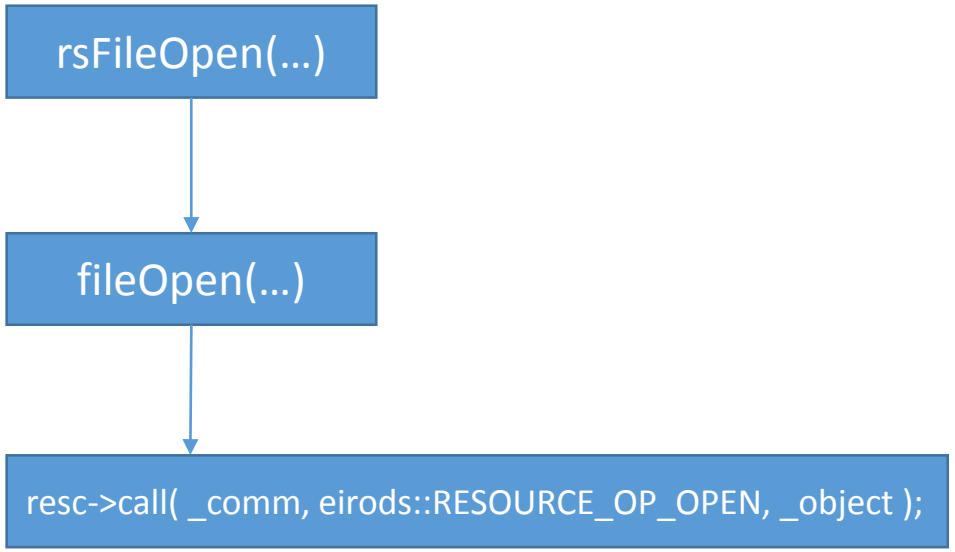


The Agent works at the most abstract levels in their hierarchies: the *plugin_base* class and *first_class_object* levels. If a *first_class_object* resolves a plugin, the system assumes the plugin can operate on that first class object.

What is a Resource Plugin? (Part 1)

- A class derived from eirods::resource_plugin which implements a POSIX-like interface to a storage medium. (Plus several other specialized operations)
- A plugin instance is created within a dynamic library via a well known factory method
- Each *Call* to a plugin routes through an *Operation* which is mapped by the plugin during the creation of the plugin in the plugin factory. These *Operations* map directly to the well known POSIX calls within iRODS.

Resource Plugin Operations



- iRODS API call delegates to the file driver interface
- fileOpen creates a *file_object*, resolves the proper *resource_plugin* and delegates to the plugin
- The plugin maps RESOURCE_OP_OPEN to whichever function the plugin developer designated as that operation within the map during plugin creation

What is a Resource Plugin? (Part 2)

What are the operations?

Open, Close, Read, Write, etc. (plus specialized operations)

Where are they defined?

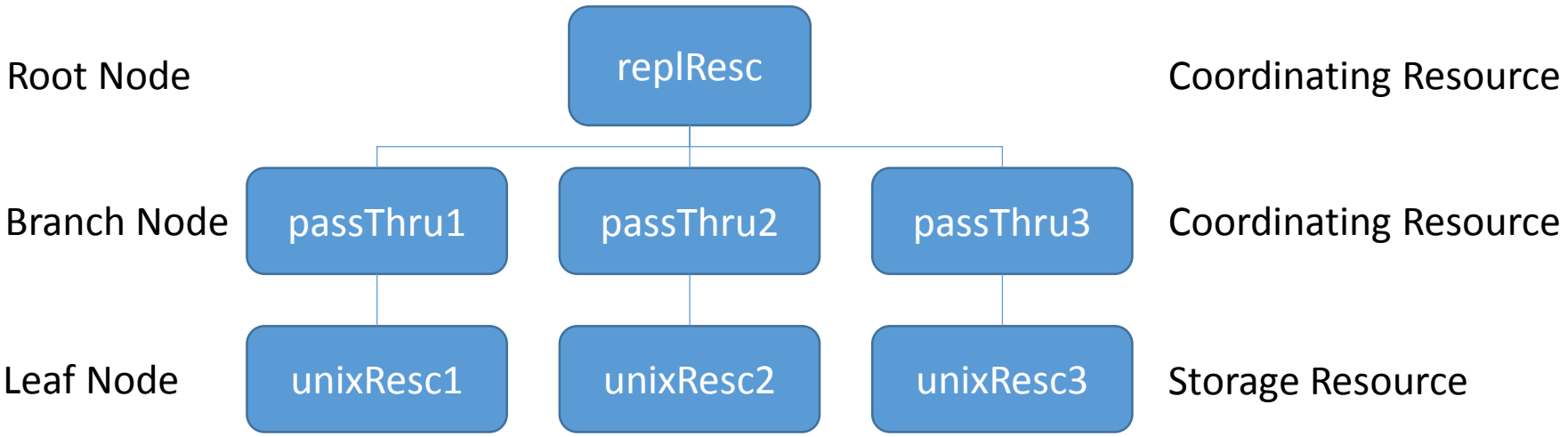
iRODS/lib/core/include/eirods_resource_constants.h

What do they map to – operations map to free functions defined within the dynamic library which are loaded at runtime and mapped within the plugin:

```
resc->add_operation( eirods::RESOURCE_OP_CREATE,      "unix_file_create_plugin" );
resc->add_operation( eirods::RESOURCE_OP_OPEN,        "unix_file_open_plugin" );
resc->add_operation( eirods::RESOURCE_OP_READ,        "unix_file_read_plugin" );
resc->add_operation( eirods::RESOURCE_OP_WRITE,       "unix_file_write_plugin" );
resc->add_operation( eirods::RESOURCE_OP_CLOSE,       "unix_file_close_plugin" );
```

Resource Composition:

Tree Terminology vs iRODS Terminology



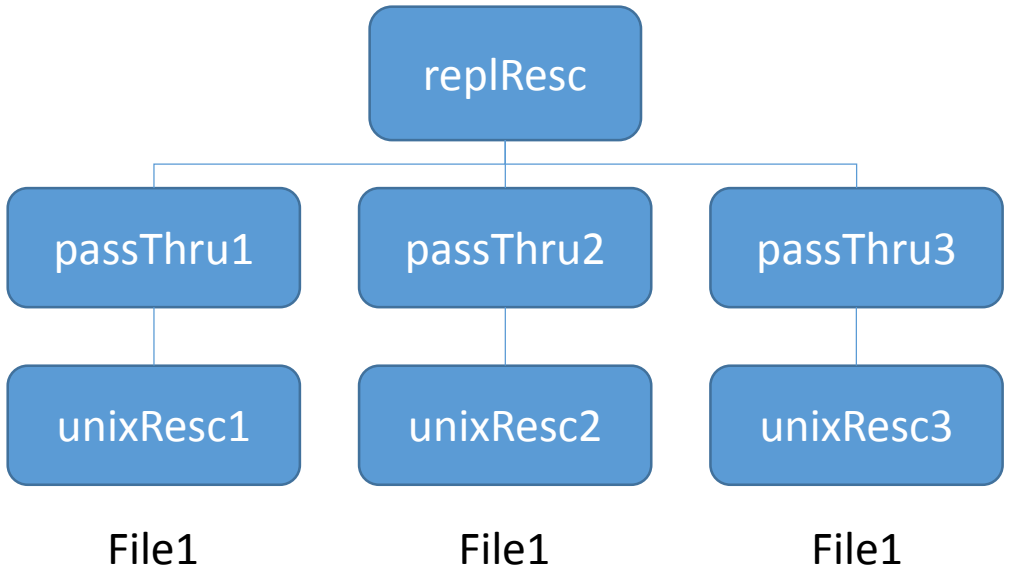
Storage Resources – reside at the physical location of the media
Coordinating Resource – purely virtual and reside at every iRODS server

Resource Composition: Hierarchy Resolution

How does the system determine which path through the tree to follow?

This can and does change based on which action is invoked: Create vs. Open with Read or Write

Solution: Everyone gets a Vote



Given an Open for Read:

- A vote is a floating point value – 0.0 to 1.0
- The root node queries its children for the action and gathers the votes.
- Each child node does the same until a leaf node is reached.
- Each leaf node examines the request and uses its own criteria for determining a vote.

  UFS e.g.,

Do I have any copy of the file?	0.25
Do I have a non dirty copy of the file?	0.5
Am I local host with a non dirty copy?	1.0

- Coordinating nodes may then do whatever they want with their children's votes. They simply must send a decision up the hierarchy.
- A path through the hierarchy is returned as coordinating nodes make decisions based on their children's votes.

Storage Resources:

- Intended to be leaf nodes – this is not directly enforced
- Only resource plugins which are tied to a particular server and reside where the storage medium is physically located
- Hierarchy Resolution – mechanism used to determine upon which data object and on which resource to operate. Votes to participate in a particular action given an iRODS data object.

  Actions include: Create, Open, and Write
- Implements POSIX *Operations* which directly access storage media

Storage Resources: The Plugin Context

```
eirods::error unix_file_redirect_plugin(
      eirods::resource_plugin_context& _ctx,
      const std::string*                _opr,
      const std::string*                _curr_host,
      eirods::hierarchy_parser*         _out_parser,
      float*                            _out_vote )
```

- Every operation provided by a plugin must start with the context as the first parameter, the rest are defined as template parameters
- The context provides:
  _ctx.fco() – The *first_class_object* in question, held as pointer to base class
  _ctx.prop_map() – the heterogeneous property map
  _ctx.comm() – the rsComm pointer for this connection
  _ctx.results() – results of dynamic policy enforcement point (advanced topic)
  _ctx.child_map() – map containing children of this node

- Note – context and available parameters can change across plugin interfaces, this is only describing the resource plugin context

Storage Resources: Hierarchy Resolution (Part 1)

```
eirods::error unix_file_redirect_plugin(
        eirods::resource_plugin_context& _ctx,
        const std::string*               _opr,
        const std::string*               _curr_host,
        eirods::hierarchy_parser*        _out_parser,
        float*                           _out_vote ) {
        eirods::error result = SUCCESS();
        // =-=-=-=-=-=-=-
        // check the context validity
        eirods::error ret = _ctx.valid< eirods::file_object >();
        if((result = ASSERT_PASS(ret, "Invalid resource context.")).ok()) {
            // =-=-=-=-=-=-=-
            // check incoming parameters
            if((result = ASSERT_ERROR(_opr && _curr_host && _out_parser && _out_vote, SYS_INVALID_INPUT_PARAM,
                                      "Invalid input parameter.")).ok()) {
                // =-=-=-=-=-=-=-
                // cast down the chain to our understood object type
                eirods::file_object_ptr file_obj = boost::dynamic_pointer_cast< eirods::file_object >( _ctx.fco() );
                // =-=-=-=-=-=-=-
                // get the name of this resource
                std::string resc_name;
                ret = _ctx.prop_map().get< std::string >( eirods::RESOURCE_NAME, resc_name );
                if((result = ASSERT_PASS(ret, "Failed in get property for name." )).ok()) {
                    // =-=-=-=-=-=-=-
                    // add ourselves to the hierarchy parser by default
                    _out_parser->add_child( resc_name );
```

Storage Resources: Hierarchy Resolution (Part 2)

```
            // =-=-=-=-=-=-=-
            // test the operation to determine which choices to make
            if( eirods::EIRODS_OPEN_OPERATION  == (*_opr) ||
                eirods::EIRODS_WRITE_OPERATION == (*_opr) ) {
                // =-=-=-=-=-=-=-
                // call redirect determination for 'get' operation
                ret = unix_file_redirect_open( _ctx.prop_map(), file_obj, resc_name, (*_curr_host), (*_out_vote) );
                result = ASSERT_PASS_MSG(ret, "Failed redirecting for open.");

            } else if( eirods::EIRODS_CREATE_OPERATION == (*_opr) ) {
                // =-=-=-=-=-=-=-
                // call redirect determination for 'create' operation
                ret = unix_file_redirect_create( _ctx.prop_map(), file_obj, resc_name, (*_curr_host), (*_out_vote)  );
                result = ASSERT_PASS_MSG(ret, "Failed redirecting for create.");
            }

            else {
                // =-=-=-=-=-=-=-
                // must have been passed a bad operation
                result = ASSERT_ERROR(false, EIRODS_INVALID_OPERATION, "Operation not supported.");
            }
        }
    }
}

    return result;

} // unix_file_redirect_plugin
```

Storage Resources: Hierarchy Resolution (Part 3)

```
// =-=-=-=-=-=-=-
   // redirect_create - code to determine redirection for create operation
   eirods::error unix_file_redirect_create(
      eirods::plugin_property_map&   _prop_map,
      eirods::file_object_ptr        _file_obj,
      const std::string&             _resc_name,
      const std::string&             _curr_host,
      float&                         _out_vote ) {
      eirods::error result = SUCCESS();
      // =-=-=-=-=-=-=-
      // determine if the resource is down
      int resc_status = 0;
      eirods::error get_ret = _prop_map.get< int >( eirods::RESOURCE_STATUS, resc_status );
      if((result = ASSERT_PASS(get_ret, "Failed to get \"status\" property." )).ok()) {
         // =-=-=-=-=-=-=-
         // if the status is down, vote no.
         if( INT_RESC_STATUS_DOWN == resc_status ) {
            _out_vote = 0.0;
            result.code( SYS_RESC_IS_DOWN );
            // result = PASS( result );
         }
```

Storage Resources: Hierarchy Resolution (Part 4)

```
    } else {
            // =-=-=-=-=-=-=-
            // get the resource host for comparison to curr host
            std::string host_name;
            get_ret = _prop_map.get< std::string >( eirods::RESOURCE_LOCATION, host_name );
            if((result = ASSERT_PASS(get_ret, "Failed to get \"location\" property." )).ok()) {

                // =-=-=-=-=-=-=-
                // vote higher if we are on the same host
                if( _curr_host == host_name ) {
                    _out_vote = 1.0;
                } else {
                    _out_vote = 0.5;
                }
            }
        }
    }
    return result;

} // unix_file_redirect_create
```

Storage Resources: Hierarchy Resolution (Part 5)

```
eirods::error unix_file_redirect_plugin(
    eirods::resource_plugin_context& _ctx,
    const std::string*                _opr,
    const std::string*                _curr_host,
    eirods::hierarchy_parser*           _out_parser,
    float*                          _out_vote ) {
    eirods::error result = SUCCESS();
    // =-=-=-=-=-=-=-
    // check the context validity
    eirods::error ret = _ctx.valid< eirods::file_object >();
    if((result = ASSERT_PASS(ret, "Invalid resource context.")).ok()) {
        // =-=-=-=-=-=-=-
        // check incoming parameters
        if((result = ASSERT_ERROR(_opr && _curr_host && _out_parser && _out_vote, SYS_INVALID_INPUT_PARAM,
                                  "Invalid input parameter.")).ok()) {
            // =-=-=-=-=-=-=-
            // cast down the chain to our understood object type
            eirods::file_object_ptr file_obj = boost::dynamic_pointer_cast< eirods::file_object >( _ctx.fco() );
            // =-=-=-=-=-=-=-
            // get the name of this resource
            std::string resc_name;
            ret = _ctx.prop_map().get< std::string >( eirods::RESOURCE_NAME, resc_name );
            if((result = ASSERT_PASS(ret, "Failed in get property for name." )).ok()) {
                // =-=-=-=-=-=-=-
                // add ourselves to the hierarchy parser by default
                _out_parser->add_child( resc_name );
```

Storage Resources: Hierarchy Resolution (Part 6)

```
eirods::error unix_file_redirect_open(
      eirods::plugin_property_map&   _prop_map,
      eirods::file_object_ptr        _file_obj,
      const std::string&             _resc_name,
      const std::string&             _curr_host,
      float&                         _out_vote )  {
      eirods::error result = SUCCESS();
      // =-=-=-=-=-=-=-
      // initially set a good default
      _out_vote = 0.0;

      // =-=-=-=-=-=-=-
      // determine if the resource is down
      int resc_status = 0;
      eirods::error get_ret = _prop_map.get< int >( eirods::RESOURCE_STATUS, resc_status );
      if((result = ASSERT_PASS(get_ret, "Failed to get \"status\" property." )).ok()) {
         // =-=-=-=-=-=-=-
         // if the status is down, vote no.
         if( INT_RESC_STATUS_DOWN != resc_status ) {
            // =-=-=-=-=-=-=-
            // get the resource host for comparison to curr host
            std::string host_name;
            get_ret = _prop_map.get< std::string >( eirods::RESOURCE_LOCATION, host_name );
            if((result = ASSERT_PASS(get_ret, "Failed to get \"location\" property." )).ok()) {
```

Storage Resources: Hierarchy Resolution (Part 7)

```
// =-=-=-=-=-=-=-
// set a flag to test if were at the curr host, if so we vote higher
bool curr_host = ( _curr_host == host_name );
// =-=-=-=-=-=-=-
// make some flags to clairify decision making
bool need_repl = ( _file_obj->repl_requested() > -1 );
// =-=-=-=-=-=-=-
// set up variables for iteration
bool         found     = false;
eirods::error final_ret = SUCCESS();
std::vector< eirods::physical_object > objs = _file_obj->replicas();
std::vector< eirods::physical_object >::iterator itr = objs.begin();
// =-=-=-=-=-=-=-
// check to see if the replica is in this resource, if one is requested
for( ; itr != objs.end(); ++itr ) {
    // =-=-=-=-=-=-=-
    // run the hier string through the parser and get the last
    // entry.
    std::string last_resc;
    eirods::hierarchy_parser parser;
    parser.set_string( itr->resc_hier() );
    parser.last_resc( last_resc );
    // =-=-=-=-=-=-=-
    // more flags to simplify decision making
    bool repl_us  = ( _file_obj->repl_requested() == itr->repl_num() );
    bool resc_us  = ( _resc_name == last_resc );
    bool is_dirty = ( itr->is_dirty() != 1 );
```

```
// =-=-=-=-=-=-=-
// success - correct resource and dont need a specific
//           replication, or the repl nums match
if( resc_us ) {
    // =-=-=-=-=-=-=-
    // if a specific replica is requested then we
    // ignore all other criteria
    if( need_repl ) {
        if( repl_us ) {
            _out_vote = 1.0;
        } else {
            // =-=-=-=-=-=-=-
            // repl requested and we are not it, vote
            // very low
            _out_vote = 0.25;
        }
    } else {
        // =-=-=-=-=-=-=-
        // if no repl is requested consider dirty flag
        if( is_dirty ) {
            // =-=-=-=-=-=-=-
            // repl is dirty, vote very low
            _out_vote = 0.25;
        }
```

Storage Resources: Hierarchy Resolution (Part 8)

```
                    } else {
                        // =-=-=-=-=-=-=-
                        // if our repl is not dirty then a local copy
                        // wins, otherwise vote middle of the road
                        if( curr_host ) {
                            _out_vote = 1.0;
                        } else {
                            _out_vote = 0.5;
                        }
                    }
                }
                found = true;
                break;
            } // if resc_us
        } // for itr
    }
} else {
    result.code( SYS_RESC_IS_DOWN );
    result = PASS( result );
}
}

return result;

} // unix_file_redirect_open
```

Storage Resources: File Open (Part 1)

```
// =-=-=-=-=-=-=-
   // interface for POSIX Open
   eirods::error unix_file_open_plugin(
      eirods::resource_plugin_context& _ctx )
   {
      eirods::error result = SUCCESS();

      // =-=-=-=-=-=-=-
      // Check the operation parameters and update the physical path
      eirods::error ret = unix_check_params_and_path( _ctx );
      if((result = ASSERT_PASS(ret, "Invalid parameters or physical path.")).ok()) {

         // =-=-=-=-=-=-=-
         // get ref to fco
         eirods::file_object_ptr fco = boost::dynamic_pointer_cast< eirods::file_object >( _ctx.fco() );

         // =-=-=-=-=-=-=-
         // handle OSX weirdness...
         int flags = fco->flags();

         // =-=-=-=-=-=-=-
         // make call to open
         errno = 0;
         int fd = open( fco->physical_path().c_str(), flags, fco->mode() );
```

Storage Resources: File Open (Part 2)

```
        // =-=-=-=-=-=-=-
        // cache status in the file object
        fco->file_descriptor( fd );

        // =-=-=-=-=-=-=-
        // did we still get an error?
        int status = UNIX_FILE_OPEN_ERR - errno;
        if (!(result = ASSERT_ERROR(fd >= 0, status, "Open error for \"%s\", errno = \"%s\", status = %d, flags = %d.",
                                    fco->physical_path().c_str(), strerror(errno), status, flags)).ok()) {
            result.code(status);
        } else {
            result.code(fd);
        }
    }

    // =-=-=-=-=-=-=-
    // declare victory!
    return result;

} // unix_file_open_plugin
```

Coordinating Resources:

- A purely virtual construct which will be instantiated at Agent startup on every server
- Provides an opportunity to influence where and how a data object may be placed, modified, or retrieved
- Directly encodes aspects of data management policy in a dynamic decision tree
- Follows the general convention of polling children, then makes a decision given the results, and then forwards the results up the tree
- Leverages the additional operations – RESOURCE_OP_NOTIFY

Coordinating Resources: Making Decisions

During the call for Hierarchy Resolution the Coordinating Resources will forward the call to their children, gather the results, and possibly use those results to make a decision.

Simple example: replication node for Open

- Forward the call to children
- Place hierarchies and votes into a map
- Sort the results
- Take the highest vote and return that vote and its hierarchy

Coordinating Resources: Talking to Children

- Children may be accessed from the plugin context via the child_map() accessor.
- This results in std::pair containing the key for the map (i.e., child name) and the value which is also a std::pair.
- This second pair is the parent-child relationship context string and a resource_ptr.

```
eirods::resource_child_map::iterator it;
float out_vote = 0.0;
for(it = _ctx.child_map().begin(); result.ok() && it != _ctx.child_map().end(); ++it) {
    eirods::hierarchy_parser parser(_parser);
    eirods::resource_ptr child = it->second.second;
    ret = child->call<const std::string*, const std::string*, eirods::hierarchy_parser*, float*>(
                _ctx.comm(), eirods::RESOURCE_OP_RESOLVE_RESC_HIER, _ctx.fco(),
                _operation, _curr_host, &parser, &out_vote);
    // do something with votes and hierarchies
}
```

Coordinating Resources: Taking Action (Part 1)

Hierarchy Resolution is the first way a resource composition is alerted to an action which is taken—RESOURCE_OP_NOTIFY is an additional *Operation* added to the plugins which provides an interface for communication directly to the hierarchy. Actions include:

EIRODS_CREATE_OPERATION

EIRODS_WRITE_OPERATION

EIRODS_OPEN_OPERATION

EIRODS_MODIFIED_OPERATION

Coordinating Resources: Taking Action (Part 2)

RESOURCE_OP_NOTIFY is called with an action of EIRODS_MODIFIED_OPERATION after the registration of a data object within the resource.

This provides a hook for the resource composition to take action as a change has happened within the hierarchy.

e.g., Replication Node will replicate a newly added or modified data object

Coordinating Resources: Rebalancing

- Initiated via a subcommand of iadmin modresc
- Provides an opportunity for a coordinating resource to make decisions about and change its current state in an administrative mode
- Expected flow is depth first—rebalance children then rebalance one's self
- Currently only implemented by the replication node—ensures all children have a replica and that all dirty replicas are refreshed