kIDL: Interface Definition Language for the Kernel

Sarah Spall
email spall@cs.utah.edu
University of Utah, School of Computing

I. Introduction

This project is part of a larger project whose goal is to decompose the Linux kernel. The Linux kernel is a shared-memory environment, and decomposing the Linux kernel provides security by confining the effects of attacks. The Lightweight Capability Domains project [6] is decomposing the Linux kernel into a share-nothing environment. In this environment, subsystems which previously existed in a shared-memory environment will now exist in their own share-nothing isolated domains. Because processes now execute in separate address spaces, functions cannot be directly called between domains. To allow these subsystems to continue to interact in this new environment, new code must be written that makes function calls across address spaces. This "glue" code marshals parameters from one address space to another, and then calls the function as if the caller was in the same address space. This code must be written for each individual function that we would like to use across domains, and this code is highly similar between functions. To automate this code generation an Interface Definition Language (IDL) is used. An IDL allows a user to describe the functions they would like to export for other domains to use, and then an IDL compiler automatically generates the glue code.

We created an IDL compiler that automatically generates this glue code. Because the Linux kernel is written in C, this IDL is targeted for the C language, and we created a new IDL syntax for describing C interfaces. Like previous IDLs' syntaxes, this new IDL syntax allows users to describe data structures, functions, and remote data structures.

From an IDL specification, two pieces of code are generated: code for the function caller, caller glue code; and code for the function provider, callee glue code. The caller glue code is used by anyone who wishes to call any of the exported functions. This glue code is called in place of the real function, and handles the marshaling of parameters across address spaces, and the remote function call using inter-domain communication (IDC). The callee side glue code is used by whomever is exporting the functions described in the IDL. This glue code intercepts the function calls meant for the callee, and then unmarshals the parameters and makes the real function call, returning the result to the caller using IDC. The code generated from the IDL is described in greater detail in section III.

We want to allow two separate domains to synchronize two separate copies of the same data, so the same piece of data can be used without passing the entire data structure every function call; only the pieces of the data structure which have changed must be passed. Since, pointers to structures are often passed during function calls, this allows the minimal amount of data to be copied between address spaces during each function call. To support this, we have created the concept of a container. A container is a structure which contains the original piece of data we would like to synchronize, as well as a reference to our local container and a reference to the remote container. These container references allow one domain to refer to another domain's data during a function call. To support this idea, the IDL syntax allows the user to specify container lifetimes with new syntax. Containers will be explained in greater detail in section III-C.

In the following sections I will explain the syntax of the IDL, the glue code which is generated, and the purpose of this glue code. The IDL compiler currently generates the caller and callee glue code functions, as well as the container structures. It also generates the extra code required to implement cross-domain function pointer calls.

II. Syntax

This section will describe the Interface Description Language's syntax.

A. Keywords

The following are keywords which can be attached to parameters and projection fields. A projection is how a C structure is described in the IDL; these will be explained in more detail in section II-C. These keywords indicate the direction of data flow during a function call as well as data structure lifetime. Here, data structure lifetime, refers to the lifetime of a "container", which is how shared memory is imitated in the share nothing environment. For variables the users wishes would persist across remote function calls, a container is allocated and saved. Containers and how they imitate shared memory will be explained in more detail in section III-C.

The **in** and **out** keywords are used to indicate the direction of data flow during a function call. This information allows the compiler to only marshal the necessary parameters during a function call. Previous IDLs [] have also used keywords to indicate the direction of data flow during a function call. The **in** keyword indicates that a parameter must be passed from the caller to the callee during the function call. When the remote function call is made the value of that parameter will be marshaled by the caller and then unmarshaled by the

callee. The **out** keyword is similarly used to indicate that the parameter needs to be passed from the callee to the caller when the remote function call returns.

In simple cases, such as a function which only accepts scalar values, this data flow may be obvious. In more complex cases, the **in** and **out** keywords can provide useful hints to the compiler about which values it is necessary to marshal. annotating functions and projections does require the user to understand the data flow of each function they wish to decompose. Labeling the fields of a projection in a discerning manner requires the user understand how the corresponding structure is accessed as it flows through function calls. This may prove to be too great of a burden for the user and useful future work would be to develop a tool which statically analyzes C data structures and the function calls which use them, to automatically generate projections.

The **alloc** and **dealloc** keywords are used to describe object lifetimes. This idea of object lifetimes creates an imitation of shared memory where the caller and callee can share an object even though they are running in a share nothing environment. Shared memory is imitated through the use of containers. A container is a structure which encloses the data the caller and callee wish to share, as well as a local and remote reference for the containers. If a container is stored by a caller or callee, this allows the data it holds to be references in a later function call. Containers will be explains in more detail in section III-C.

Alloc and **dealloc** indicate to the compiler when a container needs to be allocated and deallocated. After a container is allocated it is inserted into a cspace so it can be referenced during a later remote function call. The **alloc** keyword tells the compiler when a container should be created and inserted into the cspace. **Alloc(caller)** tells the compiler that a container should only be allocated on the caller side. Similarly, a variable is marked **alloc(callee)** if a container only needs to be created on the callee side. If a container should be allocated on both sides, then just **alloc** is used.

Dealloc is the counterpart to **alloc**, and it is used to specify that a variable's container should be removed from the cspace. Just as with **alloc**, a variable can be marked **dealloc(caller)**, to indicate it's container should be removed from the caller's cspace. A variable can be marked **dealloc(callee)**, in which case it will be removed from the callee's cspace. And if a variable's container's should be removed from both the caller's and callee's cspaces, then it can be marked just **dealloc**..

Bind tells the compiler when a container already exists for a variable. As with **alloc** and **dealloc**, **bind(caller)** means the caller has already allocated a container for this variable, and **alloc(callee)** means a container has already been allocated on the callee's side. Originally it was thought that a **bind** keyword would be unnecessary, because the compiler would simply be able to infer that a container already existed if **alloc** wasn't used. This is true if every variable which is a pointer to a projection requires a container. However, if we want to give the user the option to decide whether or not they want to reference the same data across function calls, then a bind keyword is necessary to tell the compiler when a container already exists. Otherwise, the compiler won't know if the container can be found in the caller's or callee's cspaces.

B. Simple types

Currently supported types:

- Integer data types
  - char (signed, unsigned)
  - short (signed, unsigned)
  - int (signed, unsigned)
  - long (signed, unsigned)
  - long long (signed, unsigned)
- Floating point
- Boolean
- Structures (non-circular)
- Pointer types

Currently unsupported types:

- Union types
- Enum types
- Lists

C. Structure data types

C structure types are described in the IDL with projections. A single structure may be described by many projections. This is because a projection describes more than just the fields of a structure. Just as a parameter can be marked: in, out, alloc, dealloc, and bind, so too can the field of a projection. Because the data flow and lifetime of a structure's fields can be described using a projection, a user may need to define multiple projections for the same structure.

Projections tell the compiler which fields of a structure it is necessary to marshal and unmarshal during function calls. As mentioned in section II-A the in, out, alloc, and dealloc, keywords provide hints to the compiler which fields need to be marshaled or allocated, just as for other parameters. Because there is no shared address space, when a structures is passed as an argument to a function or returned from a function call, all of the fields must be explicitly passed. The **in** and **out** keywords tell the compiler which fields need to be marshaled which directions.

Currently there are two slightly different projection forms. The first form is used for simply describing the relevant fields of a structure and any channels associated with this projection. Channels will be explained in section III-B.

```
projection <struct struct-name> proj-name {
    channel [alloc] chan1;  // optional
    type * [in,out,alloc] field1;
    type2 [in] field2;
    .
    .
    .
}
```

The second projection form allows the user to, in addition to what the previous form describes, describe channels associated with this projection that will be created elsewhere and will be provided when this projection is used.

```
projection <struct struct_name>
                                proj_name(channel arg1
                                          , ...) {
    channel [alloc] chan1; // optional
    type * [in] field1;
    type2 [out] field2;
    .
    .
    .
}
```

D. Functions

Functions are described using rpc syntax, which looks like a regular C function declaration, but with "rpc" at the beginning.

```
rpc type name(type1 [in,...]  *param1, ...);
```

This syntax is used to describe a function the user wishes to export. The return type and parameters of an rpc definition can either be one of the simple types described in section II-B or a projection. As stated in section II-A, the parameters can be marked with the keywords: in, out, alloc, dealloc, and bind, which describe which direction the values of these parameters flow during the function call, and whether or not containers should be allocated for these parameters.

From an rpc definition the compiler will generate two functions: a caller function, and a callee function. The caller function will have the same signature as the exported function, and will be used by the caller in place of the real function. The callee function will have the following function signature:

```
void name_callee();
```

This function will intercept the caller's remote function call and call the real function. The content of these functions will be explained in detail in section III-D.

E. Function pointers

The notation for describing function pointers is very similar to the notation for describing functions, and is also very similar to the C notation.

```
rpc type (*name) (type1 [in, ...] *param1, ...);
```

Function pointers are currently only supported in a projection field context.

```
projection <struct name> proj-name {
  ...
  rpc type [alloc] (*name) (type1 [in,...] *param1
                            , ...);
  ...
}
```

Function pointers can be marked with the same keywords as regular rpc parameters and projection fields. If a function pointer is marked alloc, a trampoline[1] will be allocated for the function pointer. The trampoline enables cross domain function pointer calls and will be explained in detail in section III-D.3. In addition to caller and callee functions, for each function pointer, a trampoline function is also generated by the compiler. This trampoline function is included by the caller of the function pointer

The signature of the function pointer caller's function:

```
return_type name(type1 *param1
                , ...
                , struct proj-name_container *cont
                , struct cspace *cspace);
```

The signature of the trampoline function:

```
return_type name_trampoline(type1 *param1, ...);
```

The callee function pointer glue code, has the same signature as the callee glue code for a regular function.

The trampoline function passes the extra arguments to the caller glue code function.

If a function pointer is marked **dealloc(caller)**, the copy of the trampoline code will be freed, as well as the hidden arguments structure for the function pointer. It does not make sense for the function pointer to be deallocated on the callee side, because nothing special is allocated for the function pointer. It also does not mean anything for a function pointer to be marked **in** or **out**, and as a result these will be ignored.

F. Module syntax

A module describes a collection of functions a domain wishes to provide, and the associated data structures. The module syntax is as follows:

```
module module_name (channel chnl1, ...) {
    projection <struct s> proj-name {...}
    ...
    rpc type rpc_name (type1 param1, ...);
    ...
    {
        projection <struct s> proj-name {...}
        ...
        rpc type rpc_name (type1 param1, ...);
        ...
    }
    ...
}
```

The body of a module begins with any number of type definitions, followed by any number of rpc definitions, and lastly, followed by any number of unnamed scopes. An unnamed scope introduces a new lexical scope, which makes it convenient for a user to associate type definitions with one or more rpcs. Consider the following example of how unnamed scopes are useful.

[1]Charlie Jacobsen is credited with designing how trampolines work.

```
module m (channel chan1) {

  {
    projection <struct s> p
    {
      int [in] field1;
      int [out] field2;
    }

    rpc int func1(projection p *param1);

  }

  {
    projection <struct s> p
    {
      int [out] field1;
      int [in] field2;
    }

    rpc int func2(projection p *p);
  }

}
```

In this case, two functions took the same structure as an argument, but used it in different ways. The user wanted to describe two different projections for this reason, and using unnamed scopes they were able to create two projections each with the same name.

A module also declares channels, in the form of what looks like a constructor declaration. A module declares channels for the caller to make remote function calls and for the callee to listen for remote function calls. A module must declare at least one channel. Currently, only the first channel declared is used for making and listening for remote function calls. If users wish to specify a specific channel for each rpc, a new syntax would need to be created to specify that. The channels specified in the IDL will be declared as global variables in the generated code and will be initialized by the module initialization functions. The initialization functions and other code generated for a module definition will be explained in detail in section III-A.

III. Generated Code

A. Modules

A module is used to group together the functions a domain wishes to export. These functions share a channel for making IDC calls and share a cspace for container storage. These extras, channels and cspaces, require initialization before they can be used. As a result, for each module, two module initialization functions are generated: a caller initialization function, and a callee initialization function.

The initialization functions are mainly responsible for setting up channels and cspaces. For each channel specified as part of the module, a corresponding cspace is created by the initialization function. Every channel declared as part of the module and the corresponding cspaces are declared as global variables in each the generated caller code file and the generated callee code file. Remote function calls made on a channel will use the corresponding cspace to store any containers they allocate. The channels are provided as arguments to the initialization functions, and are allocated somewhere else. These channels must be shared between the caller and callee, and the process of sharing channels happens via some other means.

Suppose someone defined the following module *vfs* in an IDL file.

```
module vfs (channel vfs_chnl) { ... }
```

In the caller glue code file, *vfs_chnl* will be declared as a global variable of a capability to a channel. This capability will be provided by whomever calls the initialization function. A corresponding cspace will also be declared as a global variable and will be allocated and instantiated in the initialization function.

The signature of the caller side initialization function is as follows.

```
int glue_vfs_init(cptr_t vfs_chnl
          ,struct lcd_sync_channel_group *_group);
```

The *lcd_sync_channel_group* stores a collection of channels to allow, in this case, the caller's dispatch loop to listen on multiple channels. The only time a caller would listen for function calls is if it exports function pointers to another domain. Therefore, in the caller's initialization function no channels are added to the *lcd_sync_channel_group*. The initialization function also creates the corresponding cspaces for each channel.

The signature of the callee's module initialization function is as the same as the caller's module initialization function. But, because the callee is providing the functions it must initialize its dispatch loop to listen on the channels it is provided with. Each channel passed to the initialization function is added to the *lcd_sync_channel_group*. The initialization function also creates the corresponding cspaces for each channel. The ideas around the dispatch loop, channels, and how they should be handled have been continually evolving, and as a result this particular part is not completely implemented.

For each module, caller and callee exit functions are also generated. The caller's exit function frees every cspace created during initialization. The caller's exit function frees every cspace created during initialization as well as removes every channels from the dispatch loop. The exit functions should only be called when the caller does not wish to use the functions in the module anymore, and the callee should only callee exit when it no longer wishes to provide the module.

The signature of the caller and callee module exit functions.

```
void glue_vfs_exit();
```

B. Channels and Structures

As shown in section II-C, projections can specify channels in two ways: either by declaring a channel as a field, or

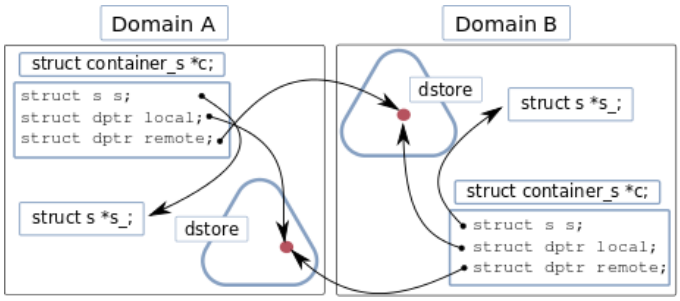

Fig. 1. Containers across domains

declaring a channel as a parameter in a projection constructor. Someone may want to specify channels for their projections if the projections contain function pointers and they want to control which channel is used for the remote function call.

This is how a channel is specified as a projection field.

```
projection <struct fs> fs {
  channel [] chnl1;
  .
  .
  .
}
```

The container structure corresponding to the *fs* projection will have a channel field. If *chnl1* was marked **alloc** then a channel would be allocated and stored in the container.

This is how a channel is specified as a parameter of a projection constructor.

```
projection<struct fs> fs(channel chnl1) {
  .
  .
  .
}
```

In this case, the *fs* projection expects to have a channel passed to it when it is used. This projection *fs*'s corresponding container will have a channel field, which would be set to the channel *fs* receives when it is initialized. Who provides this channel will depend on where the projection is referenced.

C. Containers

Containers are structures created by the compiler that contain extra information, which allows us to create an imitation of shared memory. A container will have a minimum of three fields: a field for the piece of data we would like to share, a local reference field, and a remote reference field. A container may also hold additional information in the form of channels. A container's references allow us to refer to the other domain's copy when a function call is made, by passing a reference with the other parameters. Refer to figure 1 for a graphical depiction of how two cross-domain containers are linked.

A container has two reference fields: a local reference, and a remote reference. During a function call, where both the caller and callee have already allocated containers for a particular variable, the caller will pass its remote reference to the callee. This allows the callee to locate its corresponding container in its cspace. In the case where both containers have not been allocated yet, both the caller and callee will pass their local references, so the other side can store it for later use.

Containers could be used for all parameters and return values that are pointers, however, this doesn't provide any benefit in the case of simple data types. For simple data types, such as int, a container could be created, but no portion of the data will remain the same when it is updated. It costs the same to send a remote reference from one domain to the other as it does to send the value. In the case of a data structure maybe only one or two fields will be changed at a time. Therefore, currently containers are only created for pointers to structure data types.

A container for a structure takes the following form:

```
struct structure_container {
  structure real;
  cptr my_ref;
  cptr other_ref;
  cptr chnl1; // optional
}
```

The *my_ref* field holds the position in the container's domain's cspace that the container is stored at. The *other_ref* field stores the position in the other domain's cspace the corresponding container resides at. The optional *chnl1* field stores a channel declared by this container's corresponding projection definition.

In summary, containers are used to link two pieces of data across two domains. This can be useful if two domains are continually modifying the fields of a structure, but never all at once. Using containers, two domains can synchronize their changes without needing to pass the entire structure every time a field is updated.

D. Functions

For every rpc definition in an IDL file two functions are generated: a caller function, and a callee function. The caller function is included by the caller of the real function, and handles the marshaling of parameters and making the remote function call. The callee function is included by whomever is providing the real function, and handles the unmarshaling of parameters and makes the real function call.

Suppose one domain wishes to export the following function.

```
int register_fs(struct fs *fs);
```

In the IDL file this function would be written as an rpc that looks like this:

```
projection <struct fs> fs {
    int [in] id;
    int [in, out] size;
}

rpc int register_fs(projection fs
                    [alloc(callee)] *fs);
```

From these definitions, the IDL compiler generates two functions: the caller glue code function and the callee glue code function. In the next section I will explain the caller function.

1) Caller function: The generated caller function will have the same function signature as the original *register_fs* function, but will handle marshaling parameters and making the remote function call. In this section I will detail everything that occurs in a generated caller function.

First, for every parameter which is a pointer, a projection, and is marked **alloc** or **bind** it is necessary to allocate a new container or locate the existing container. If the container has already been allocated, which in this case it has, the caller side code uses the *container_of*[2] function to access the existing container; *container_of* does math to calculate the position of the container from the parameter.

```
struct fs_container *fs_container
                = container_of(fs
                               ,struct fs_container
                               ,fs);
```

A parameter's corresponding container must be accessed every time the parameter is used so a remote reference can be passed to the callee. This allows the callee to access their own copy of the container.

If any of the *register_fs* rpc's parameters referred to a projection that declared a channel those would be allocated or initialized here. This is so these channels can be marshaled to the caller.

All parameters marked **in** and the necessary container or channel references are marshaled in preparation for the remote function call. In this example we marshal the *fs_container*'s local reference because the *register_fs* rpc's *fs* parameter is marked **alloc(callee)**. The callee code will allocate its own container and store the caller's local reference for later use. Projection *fs*'s two fields *id* and *size* are also being marshaled because they were both labeled **in**.

```
lcd_set_r1(fs_container->my_ref.cptr);
lcd_set_r3(fs->id);
lcd_set_r4(fs->size);
```

Register 0 is reserved for the function tag, which identifies which function is being called. Here we set this register to the function tag for *register_fs*.

```
lcd_set_r0(1);
```

[2]The container_of function was implemented by Charlie Jacobsen.

After marshaling is finished, the remote function call is made. This is done using a synchronous IDC call on a channel specified by the module containing the *register_fs* rpc. In this case the channel is *vfs_chnl*.

```
ret = lcd_sync_call(vfs_chnl);
```

After the remote function call returns, the return value and all parameters labeled **out** are unmarshaled. If the callee allocated any containers, the caller will unmarshal those references and store them in the corresponding container. In this example the caller received a reference from the callee and is storing it in the *fs_container*'s other reference field. The *fs* projection's field *size* was marked **out**, so we unmarshal its value.

```
fs_container->other_ref = __cptr(lcd_r1());
fs_container->fs.size = lcd_r2();
return lcd_r3();
```

These are the major portions of the generated caller function. In the next section I will explain the generated callee function.

2) Callee function: This generated callee function will handle the unmarshaling of parameters, will make the real function call, and will return the appropriate values to the caller. In this section I will detail the contents of the callee function.

The generated callee function will have the following signature:

```
void register_fs_callee();
```

First, each of *register_fs*'s parameters must be declared. If a parameter has a container, the container is declared instead of the parameter. In this example *register_fs*'s *fs* parameter has a parameter, so we declare it here.

```
fs_container *fs_container;
```

Fs was marked **alloc(callee)**, so the callee code will allocate this container and insert it into the cspace. This is also where regular parameters would be allocated, if any were declared.

```
fs_container = kzalloc(sizeof(*fs_container)
                       , GFP_KERNEL);

ret = vfs_cap_insert_fs_type(minix_cspace
        ,fs_container
              ,&fs_container->my_ref);
```

If *fs* had been marked **bind** rather than **alloc**, *fs_container* would have been looked up in the cspace using the remote reference received from the caller.

After every parameter has been allocated, and every container has been either allocated or looked up in the cspace, the parameter's values are unmarshaled. In this example, the

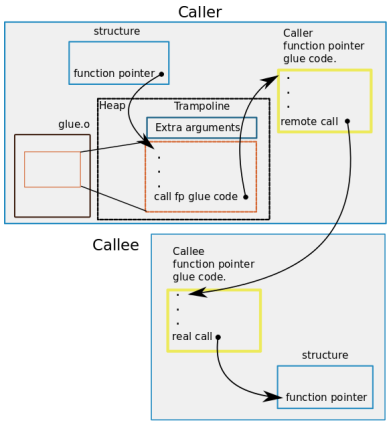

Fig. 2. Internal organization of trampoline code

callee sets it's *fs_container*'s remote reference to the caller's *fs_container*'s local reference. If the callee had looked up *fs_container* in the cspace, this reference would have been unmarshaled previously. The *fs_container* fields *id* and *size* are unmarshaled as well because both were marked **in**.

```
fs_container->other_ref = __cptr(lcd_r1());
fs_container->fs.id = lcd_r3();
fs_container->fs.size = lcd_r4();
```

After the parameters are unmarshaled, the real function call is made.

```
register_fs_ret = register_fs(&fs_container->fs);
```

All parameters marked **out** are marshaled to be sent back to the caller. In this case *fs_container*'s only field marked out was *size* and it is marshaled here. In addition to marshaling the parameter values, because the parameter *fs* was marked **alloc(callee)** the callee allocated its own container and the callee's local reference for this container must be passed back to the caller. The synchronous reply call is then made.

```
lcd_set_r1(cptr_val(fs_container->my_ref));
lcd_set_r3(fs_container->fs.size);
ret = lcd_sync_reply();
```

This is a basic example of what a generated callee function consists of. In more complicated cases, the callee function may also allocate function pointer trampolines to enable cross-domain function pointer calls. The code which makes this possible will be explained in the next section.

3) Function pointers and trampolines: If any function pointers are declared in the IDL file, then caller and callee functions will be generated for them. The function pointer caller function has the following signature:

```
return_type name(type1 *param1
                 , ...
                 , struct proj_name_container *cont
                 , struct cspace *cspace);
```

The last two arguments are extra arguments which must be passed to the callee so the function pointer can be located and called. There is an additional piece of code which is generated for each function pointer called a "trampoline" function. This trampoline function will be installed in place of the real function pointer on the caller's side, and will call the function pointer's caller function with the extra arguments. The following trampoline function will be generated:

```
LCD_TRAMPOLINE_DATA(new_file_trampoline);
int
LCD_TRAMPOLINE_LINKAGE(new_file_trampoline)
name_trampoline(type1 *param1, ...)
{
 int (*volatile namep)(type1*
                       , ...
                       ,struct name_struct_container*
                       ,struct cspace*);

 struct name_hidden_args *hidden_args;

 LCD_TRAMPOLINE_PROLOGUE(hidden_args
                         ,name_trampoline);

 namep = name;

 return namep(param1
              , ...
         , hidden_args->name_struct_container
         , hidden_args->cspace);
}
```

In order for one domain to use the function pointers provided by another, the caller must install a pointer to the trampoline function in place of the real function pointer. The rest of this section will explain how this is done.

If any of a function's parameters are structures which contain function pointers, then special code may be generated which enables cross-domain function pointer calls. This involves declaring a "hidden arguments" structure for each function pointer; the "hidden arguments" structure contains the extra arguments mentioned previously. Then the trampoline function generated for the function pointer is copied onto the heap, so it can be installed in place of the real function pointer.

```
struct name_hidden_args *name_hidden_args
          = kzalloc(sizeof(*name_hidden_args)
                    , GFP_KERNEL);
```

The trampoline code generated for the function pointer is copied onto the heap and then stored in the hidden arguments struct. This is so the function pointers trampoline function can access the extra arguments and call the caller function.

```
name_hidden_args->t_handle
        = LCD_DUP_TRAMPOLINE(name_trampoline);
```

In addition to copying and storing the trampoline code into the hidden arguments structure, the extra arguments fields of the hidden structure must also be initialized. On line 4 in the next code excerpt, the hidden argument which stores the container for the structure that has the real function pointer, is set. And on line 7, The cspace where the container can be

found is set. And finally, on line 9, the trampoline is installed in place of the real function pointer.

```
name_hidden_args->t_handle->hidden_args
                    = name_hidden_args;

name_hidden_args->name_struct_container
                  = name_struct_container;

name_hidden_args->cspace = cspace;

name_struct_container->name_struct.name
 = LCD_HANDLE_TO_TRAMPOLINE(
      name_hidden_args->t_handle);
```

This code is only necessary for the domain which wishes to call the function pointer. The domain providing the function pointer can simply include the generated callee file like it would with a regularly exported function.

IV. Related Work

Common Object Request Broker Architecture (CORBA) provides a specification for doing distributed object-oriented computing. CORBA specifies an Interface Definition (description) Language to describe object interfaces [9]. Various mappings, such as CORBA IDL to C++, define how the IDL is mapped to the target language. THE CORBA IDL compiler generates client and server glue code, which handles marshaling and cross-network function calls. This IDL syntax allows a user to describe an Object and the functions that may be performed on that object.

Other microkernel based projects such as L4Ka::Pistachio [4] and Barrelfish [2] also use Interface Description Languages to generate communication glue code. L4Ka::Pistachio has an interface description language, IDL4, to generate its stub code. IDL4 is written in C++ and supports mappings from the CORBA IDL and DCE IDL [5] to the C language for the Pistachio, Hazelnut, and Fiasco L4 microkernels.

Barrelfish is a multikernel, where multiple OS instances exist in separate domains and communicate using explicit messages [2]. Flounder [1] is the Interface Description Language for the Barrelfish project.

Also created in association with the Barrelfish project is Filet-o-Fish (FoF), a tool for developing Domain Specific Languages (DSLs) for operating system development [3]. FoF is an "embedding' of C in Haskell. To implement a DSL using Filet-o-Fish, the developer implements a parser, and a backend which takes the AST produced by the parser, and produces FoF code by composing the FoF constructs. The FoF compiler then translates these constructs into C code. The FoF paper mentions many difficulties that I myself ran into, and I wish I had known about Filet-o-Fish when I began this project. The most difficult part of this project was translating the AST produced by the parser into a C AST. Having a higher-order combinatorial representation of C constructs likely would have made this project easier to implement.

Other related work is Google protocol buffers [8] and Cap'n Proto [7]. Google protocol buffers serializes structured data so it can be sent across "the wire" and retrieved on the other end. Using Google protocol buffers involves writing a *.proto* file which contains a description of the data structure the user wishes to serialize. The compiler then creates a class that implements automatic encoding and decoding of the data; Google protocol buffers does not support C. This can be compared to how this project chose to "serialize" C structures by describing them using projections and rather than serializing them in the form of a string, marshals each field individually and reconstructs the structure on the other side.

Cap'n Proto is another serialization project which was spun off from Google protocol buffers. Unlike Google protocol buffers, there is an implementation for serialization in C. Maybe this could be used as an alternative to the current system of marshaling data.

V. Future Work

Useful future work on this project could be to develop a tool which uses static analysis to automatically generate projection definitions. This could be very useful because it is likely a great burden on the user to determine all of the necessary projections for anything but a very simple interface. Another piece of work, which is more necessary than automatically generating projections, is adding a sequence type to the IDL. It is a major limitation that lists are not currently supported and would be important to add in the future.

A sequence type could represent lists of an unknown length, as well as of a known length. Alternatively two different types could be created, one for lists of a known length and another for lists of an unknown length. Adding these two types would involve create new classes, as well as adding support to the parser. In the case of a sequence type for lists of unknown length, support would need to be added to determine how to marshal a sequence at runtime. This is because the current system of assigning registers for marshaling at compile time, would not work for lists of an unknown length.

There are other C types which are not supported, such as unions and enums. Support for these could also be added to the IDL.

VI. Conclusion

In order to decompose the Linux kernel, extra code must be written that enables function calls, which previously occured in a shared address space, to occur across address spaces. This code must be be written for every function we wish to call across address spaces, and it is highly similar for each function. For this project, we created an IDL compiler which automatically generates this code. In addition to allowing users to describe data structures and functions as previous IDLs have, users can also associate to pieces of data across two separate address spaces. This is implemented using containers, which can relate two pieces of data. Using containers, the same pieces of data can be referred to across

many remote function calls, in some cases reduing the need to copy entire data structures between address spaces.

Currently the code which generates caller and callee glue code for functions is complete. As well as the code which enables making remote function pointer calls. The code which initializes modules is partially complete and I am in the process of finishing this code.

VII. Acknowledgment

I would like to thank John Regehr, Matthew Flatt, and Matt Might for agreeing to be on my Master's project committee. I would also like to thank Anton Burtsev for allowing me to work on this project. The research presented in this report was supported by the National Science Foundation under the Grants No. 1319076 and No. 1527526.

References

[1] Baumann, A. Inter-dispatcher communication in barrelfish. Tech. rep., Barrelfish Technical Note, 2010.

[2] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 29–44.

[3] Dagand, P.-E., Baumann, A., and Roscoe, T. Filet-o-fish: practical and dependable domain-specific languages for os development. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 35–39.

[4] Group, S. A. L4ka::pistachio microkernel.

[5] Group, T. O. Cde 1.1: Remote procedure call, 1997.

[6] Jacobsen, C., Khole, M., Spall, S., Bauer, S., and Burtsev, A. Lightweight capability domains: towards decomposing the linux kernel. *ACM SIGOPS Operating Systems Review 49*, 2 (2016), 44–50.

[7] Varda, K. Cap'n proto.

[8] Varda, K. Protocol buffers: Googles data interchange format. *Google Open Source Blog, Available at least as early as Jul* (2008).

[9] Yang, Z., and Duddy, K. Distributed object computing with corba. Tech. rep., DSTC Technical Report 23, 1995.

Appendix

This appendix will detail the implementation details of the compiler. Writing this compiler was a learning process for me and in some situations the best design choices may not have been made. I will point out potential problems with these choices and suggest alternatives for them. I will also explain ways to extend the compiler with new features, such as dispatch loop generation. I will first note that C++ was likely not the best choice for this project, a better choice would have been Racket. Doing code generation in Racket is likely much easier than doing code generation in C++.

A. Front end

The compiler uses a parser generator; likely any parser generator could be used. The parser generator requires a grammar file, which is called *lcd_idl.peg* and is in the *parser* directory. This file describes the syntax of the IDL and how it is parsed into a tree.

I will now describe the front end tree an IDL file is parsed into. Definitions for these classes are in *lcd_ast.h* in the *include* directory and implementation for these classes is found in the *ast* directory. There are 6 important classes: Type class, Variable class, Rpc class, Module class, Project class, and LexicalScope.

A Project is made up of multiple modules as well as a top level scope which contains type definitions used by every module. At this point, development has focused solely on individual modules and not how multiple modules may interact. The implementation of the *Project* class can be found in *lcd_ast.cpp* in the *ast* directory.

Each module defined in an IDL file is parsed into a new *Module* instance. A Module consists of: *Rpc* definitions, a *LexicalScope*, channels, and cspaces. The implementation of the *Module* class is found in *lcd_ast.cpp*.

The *LexicalScope* class describes a scope for type definitions, variables, and identifiers. A new scope is introduced at these times during parsing: when a new interface is parsed, a projection constructor is parsed, an unnamed scope is parsed, or an rpc is parsed. A new scope is introduced at each of these times because new types may be introduced or new variables are introduced. The scope keeps track of variables for two reasons: first, so channel variables can be looked up in the scope and used to initialize projection constructors; and second, to keep track of which identifiers have been used.

Each rpc defined in an IDL file is parsed into a new *Rpc* instance. In addition, an *Rpc* will be created for each function pointer found in the IDL file. The rpc represents a function being provided from one domain to others. The implementation for the *Rpc* class can also be found in the *lcd_ast.cpp* file.

There is an abstract class *Variable* which is inherited from to implement the various types of variables. The various variables are: *GlobalVariable*, *Parameter*, *ReturnVariable*, and *ProjectionField*. I think having a variable, a global variable, and a return variable class may be unnecessary and these could be collapsed into just a single class. But, as it is currently I will talk about the different uses for each of these variable types.

- ***GlobalVariable*** is used for the channels specified by a module in the IDL file. These channels are declared as global variables in the output C files. *GlobalVariable*s are not marshaled between domains and therefore cannot be set as in, out, alloc, dealloc, or bind as other variables can be.
- ***Parameter*** represents an rpc's parameter. Parameters can be marked as in, out, alloc, dealloc, and bind, which tell the compiler how to generate code for the parameter.
- ***ReturnVariable*** represents the return value of an rpc. The return value of an rpc is represented as a variable so it can be assigned a name and easily declared in the generated glue code for the function.
- ***ProjectionField*** is a field in a projection. The reason this is treated as a variable is because I wasn't sure what structure fields are usually considered to be. And it fit nicely in with generating the C AST.

The goal is to support C types such as Integers and structures. The organization of the supported types is as follows. There is an abstract *Type* class and all types inherit from this class. The classes, which inherit from *Type* are: *InitializeType*, *UnresolvedType*, *Function*, *Typedef*, *Channel*,

VoidType, *IntegerType*, *ProjectionType*, and *ProjectionConstructorType*. The two most unusual of these classes is InitializeType and UnresolvedType; these do not represent types as much as future computations to calculate a type. In the beginning, Pointer was treated as a type just as integer is. I changed this to make pointer a property of a variable rather than a type, because it fit in better with the C tree and generating C code. I am not sure if this was the correct approach, however it fits in well with the implementation.

- The ***UnresolvedType*** class is used as a placeholder when the parser encounters a type whose definition has not been resolved yet. After parsing is complete, the unresolved type will resolve to the real type if possible. If the type cannot be resolved after parsing it is an error.
- The ***InitializeType*** class is used to initialize instances of *ProjectionConstructorType* after parsing is complete. A usage of a projection constructor cannot be initialized until after parsing because a field of the projection may be unresolved, and before a projection constructor can be initialized the type must be copied. In order to initialize projection constructor types after parsing, it is necessary to save the information of what we want to initialize the ProjectionConstructorType with. The IDL was not originally intended to support this sort of variable passing. As a result, I am not sure if this was the best way to implement this feature.
- The ***IntegerType*** class represents the C integer data types: char, short, int, long, and long long. Rather than creating a different class for each integer data type, I decided to create one class and use an enum field to distinguishes them. I do not know if this is better or worse than each integer type having its own class.
- The ***Typedef*** class represents a C typedef of the form:

```
typedef unsigned int size_t;
```

  Adding a struct typedef form would be as easy as adding the form to the parser. The typedef class is merely a wrapper around another type.
- The ***Function*** class represents the type of a C function. An instance of this class is created each time a function pointer is encountered by the parser, either as an rpc parameter or as a projection field. After parsing, all functions are converted into rpcs; this is to re-use the existing code which generates caller and callee functions for rpcs.
- The ***VoidType*** class represents the void type in C.
- The ***Channel*** class represents the LCD notion of a channel, although I am not sure this type is actually used. I believe every time a channel is created in the parser, a cptr_t projection is returned rather than an instance of the *Channel* class.
- The ***ProjectionType*** class represent the first of two notions of a projection. A projection describes a C structure and the relevant fields it is necessary to marshal. A projection type also specifies any channels that it will allocate. These channels may be used for making function pointer calls, or may be passed to a projection constructor type.
- The ***ProjectionConstructorType*** class represents the second notion of a projection; a projection that has parameters that must be initialized when it is used. When a projection constructor type is referred to in an IDL file, arguments must be provided. These parameters are only allowed to be channels, and these channels are used for the same reason as in a regular projection type. After parsing, the projection constructor's parameters are initialized with the arguments provided from a usage of a projection constructor type.

```
projection p1 *field1(chn1, chn2);
```

Whenever a usage of a projection constructor type is encountered by the parser, the type is wrapped in an *InitializeType* and the arguments are given to the *InitializeType*. After parsing, the projection constructor will be initialized with the arguments.

One glaring omission is lists. Although, for instance, char* is supported, it is assumed that it is a pointer to a single char not a sequence of chars. This is clearly a major limitation and can be remedied with the addition of a sequence type. This type could represent lists of unknown length. Because a list's length may not be known at compile time, additional support will be required in the form of a runtime function for marshaling sequences.

B. Tree transformations

After parsing, additional transformations are made to the tree produced by the parser. Each transformation begins at the project level, and is propagated to every module in the project. Implementing a new transformation is done by adding a new function to the Project and Module classes; additional functions are added depending on the transformation. Every transformation is called from the main function. Here the current transformations are detailed in the order which they occur.

- ***trampoline structure creation***: For each function pointer encountered, a hidden arguments structure is created. This structure contains extra arguments to be passed when the function pointer is called. At the end of this transformation a projection for each function will have been created and inserted into the appropriate scope.
- ***function pointer to rpc***: For each function pointer encountered a new rpc is created. When the glue code is being generated for each rpc, it is checked whether it is a regular rpc or an rpc created from a function pointer. In the case it is an rpc created from a function pointer, slightly different code will be produced.
- ***function tag generation***: Every rpc is assigned a unique tag, an integer, which is used to identify which rpc is being called.
- ***resolve unresolved types***: Each usage of a type which is unresolved after parsing is now resolved. If the type

still cannot be resolved then it is an error. If the type is successfully resolved it is installed in place of the unresolved type. In some cases it may not matter which order these transformations occur in, however, it does matter that this transformation occurs before the type copying transformation.

- ***creation of container variables***: A container variable is created for each variable that requires a container. Currently a variable requires a container if it is: a pointer, a projection variable, and is marked bind or alloc in the IDL.
- ***type copying***: All usages of a type point to the same type object. This is useful for resolving the unresolved fields of a projection, because, the fields only need to be resolved once, rather than for each usage of the projection. However, this also causes problems because the fields of a projection have an accessor field, which for different instances of the type should refer to different variables. In order for these accessors to be set, the projection type must be copied. This choice to allow projection fields to have accessors is very useful when generating the C code that accesses variables and fields. The copying of types is also required before initializing usages of projection constructor types.
- ***initialize projection constructors***: Projection constructor types must be initialized with channels. The type copying phase must occur before this phase so each usage of a projection constructor can be initialized with different channels. To initialize a projection constructor type, the channels saved in the *InitializeType* are passed to the projection.
- ***set accessors***: Variables have an accessor field, which points to the variable that "accesses" it. This is mostly useful for projection fields. The accessors for every variable are set so when the C code is generated, the access of each variable is generated properly. This must occur after copying of types happens, because this modifies projection fields.
- ***prepare for marshaling***: Each parameter and return value of each rpc is assigned a register. Currently, it is possible to assign registers at compile time because lists are not supported. If a sequence type is added to support lists of unknown length, variables of type sequence could not be assigned registers at compile time. A sequence type would require registers for marshaling to be determined at runtime.

These are the current transformations that occur after parsing. It is easier to reason about the small changes each individual pass makes rather than worrying about all of them combined.

C. Backend

After the IDL file is parsed into the intermediate representation and the tree is transformed, the resulting tree is passed to the code which generates the glue code. The result is C code in the form of a C AST. The definitions for this C AST can be found in *include/ccst.h*. Each of these classes has a write function which takes a file and writes itself to that file. The constructors and write functions for each class in *include/ccst.h* can be found in *ast/ccst.cpp*.

In general, there are two files we would like to generate: a caller glue code file, and a callee glue code file. Caller and callee are referred to, respectively, as client and server in the compiler. All of the code which generates the C AST is in the *code_gen* directory. First I will explain the generation of the client side glue code. Then I will explain the generation of the server side glue code.

The code which does the majority of client file generation is in *client.cpp* in *code_gen*. A single client file is generated for a module. There is a top level function *generate_client_source* which accepts a Module and returns the C AST for the file. Here I will describe the important parts of the client glue code file.

- Declare the channels specified as a part of the module in the IDL file. These channels are declared as global variables and will be initialized by the module initialization function.
- Declare the cspaces associated with each channel specified as a part of the module in the IDL file. These cspaces are declared as global variables and will be initialized by the module initialization function.
- Define a module initialization function, which initializes the channels and cspaces declared as global variables.
- Define a module exit function, which frees every cspace allocated in the initialization function and performs other exit duties.
- Define any structures which were created by the compiler and used in the caller glue code.
- For each regular rpc defined in the IDL file, generate a caller glue code function. Caller glue code functions will be explained in more detail later.
- For each rpc defined by a function pointer, generate a callee glue code function. This is because it is currently assumed that only the caller exports function pointers to the callee. This clearly will not always be true, and the caller and callee glue code for function pointers could easily be put in its own file.

The generation of server files is mainly done in *server.cpp* in *code_gen*. Just as a single client file is generated for a module, so is a single server file. There is a top level function *generate_server_source* which accepts a Module and returns the C AST for the file. Here I will describe the important parts of the server glue code file.

- Declare the channels specified as a part of the module in the IDL file. These channels are declared as global variables and will be initialized by the module initialization function.
- Declare the cspaces associated with each channel specified as a part of the module in the IDL file. These cspaces are declared as global variables and will be initialized by the module initialization function.
- Define a module initialization function, which initializes the channels and cspaces declared as global variables. This function is slightly different than the caller side

initialization function; it also inserts the channels into a dispatch loop so it can listen for function calls on these channels.
- Define a module exit function, which deallocates all of the cspaces the initialization function allocated, and removes the channels from the dispatch loop.
- Define a callee glue code function for each regularly defined rpc. The contents of the callee side glue code functions will be explained in depth later.
- Define a caller glue code function for each rpc defined by a function pointer. As stated previously, it is currently assumed that only the caller exports function pointers to the callee. These function pointer caller glue code functions could be placed in their own file, so they could be used by whomever is calling them.

Most of the complexity lies in the code that generates the caller, glue code function body. This code is mainly in the *caller_body* function in *client.cpp*. This function returns a compound statement, which is a combination of declarations and statements, where all declarations occur before statements. Here I will explain the pieces of a caller glue code function.

- For each parameter which has a container, declare the container. Then allocate each container if it does not already exist, otherwise access the existing container. If the container needs to be inserted into a cspace, this is how the correct cspace is determined. If this is the caller function for a regular rpc, then choose the cspace associated with the first channel specified for the module the rpc is in. If this is an rpc defined by a function pointer, use the cspace associated with the first channel specified by its container. If its container doesn't specify any channels, use the same cspace which would be used for a regular rpc.
- For each parameter which is a projection and specified in the IDL that it wants to allocate its own channels, allocate all of the channels it specified. All allocated channels are stored in the projection variable's corresponding container.
- For each parameter which is a projection constructor and specified in the IDL that it must be initialized with channels, initialize each channel. As with the allocated channels, these channels are also stored in the projection variable's corresponding container.
- After everything that needs to be allocated and initialized is, we marshal all of the parameters which are marked "in" and their corresponding container references, if they have a container. If the rpc we are generating a caller body for was defined by a function pointer, then it is also necessary to marshal the function pointer's hidden arguments. The rpc's unique tag is also marshaled, always in register 0, so the function being called an be identified by the callee.
- Now the remote call is made using the first channel specified by the module, or if this rpc was defined by a function pointer, using the first channel specified by the function pointer's container. If there is no channel specified by the container, the function pointer call is made using the same channel a regular rpc would be called on. This is a synchronous call.
- When the IDC call returns, all of the parameter and container fields marked "out" are saved. If the return type of the function is not void, a variable is declared and allocated for the return value, and is then populated. If this is an rpc defined from a function pointer, then the function pointer's hidden arguments which are marked "out" are also unmarshaled.
- At this point, before we return to the caller, if any parameters are marked dealloc, we deallocate them. Only parameter's which have a container can be deallocated. Depending on whether the parameter is marked *dealloc(caller)* or *dealloc(callee)* determines how complicated deallocating is. For instance, if the parameter is marked *dealloc(callee)* then the caller glue code will merely remove the container from the cspace. If the parameter is marked *dealloc(caller)* then the container is removed from the cspace, and all channels associated with the container, which are marked *dealloc(caller)*, are deleted. If the projection we are deallocating had function pointers in it, we also deallocating the hidden arguments and trampoline code for these function pointers. Finally, the container itself is freed.
- Finally, we are ready to return to the caller.

Just as when generating client side glue code, most of the complexity of generating a server side glue code lies in the code that generates the server glue code function body. The function which generates the rpc callee function bodies is *callee_body* in *server.cpp*. This function returns a compound statement, which is a combination of declarations and statements, where all declarations occur before statements. Here I will explain the pieces of a callee glue code function.

- First, we declare each parameter and parameter's container, if it has one. If the rpc we are generating callee glue code for was defined by a function pointer, then we also need to declare the hidden arguments.
- For each parameter's container, either allocate the container, or lookup the existing container in the cspace. Container's are looked up using the container reference the callee received from the caller. If this rpc was defined by a function pointer, then the hidden arguments will also need to be allocated. Regular parameters, with no containers will also need to be allocated.
- For each parameter which is a projection with function pointers, we perform a setup so the callee, can call these function pointers. As stated previously it is assumed that the caller exports function pointers to the callee. This setup involves installing a pointer to the function pointer's "trampoline" function in place of a pointer to the real function. To do this, first a hidden arguments structure is declared for each function pointer which was marked *alloc(callee)* in the IDL file. Then, each of these hidden arguments structures is allocated. Then

the trampoline code for the function pointer is copied to the heap and installed in the hidden arguments structure. Then various field linking is done; this field linking can be more easily understood from looking at the produced code. The code which allocates hidden arguments structures can be found in *trampolines.cpp* in the *code_gen* directory.

- After all of the parameters, containers, and hidden argument structures have been declared and allocated, then the parameters which were marked *in* are unmarshaled. At this point, the container references have already been unmarshaled, because they may have been required to lookup a container in the cspace.
- Now, the real function call is made. If the rpc was function pointer defined, then the call is made by accessing the function pointer stored in the hidden argument container of the projection which contained the function pointer.
- After the function call returns is when any parameters which were marked *dealloc* are deallocated. Deallocation on the server side follows the same process as on the caller side, except the simple case of only removing a container from the cspace occurs if the parameter was marked *dealloc(caller)* and the complex deallocation case occurs if the parameter was marked *dealloc(callee)*.
- After the function call returns, all parameters and container references marked *out* are marshaled back to the caller, as well as any non-void return value.
- The IDC reply call is then made.