Dynamic RDMA Credentials

James Shimek, James Swaro
Cray Inc.
Saint Paul, MN USA
{*jshimek,jswaro*}*@cray.com*

***Abstract*—*Dynamic RDMA Credentials* (DRC) is a new system service to allow shared network access between different user applications. DRC allows user applications to request managed network credentials, which can be shared with other users, groups or jobs. Access to a credential is governed by the application and DRC to provide authorized and protected sharing of network access between applications. DRC extends the existing protection domain functionality provided by ALPS without exposing application data to unauthorized applications. DRC can also be used with other batch systems such as SLURM, without any loss of functionality.**

In this paper, we will show how DRC works, how to use DRC and demonstrate various examples of how DRC can be used. Additionally, future work for DRC will be discussed, including optimizations for performance and authorization features.

***Keywords*-Cray; DRC; Cray XC; Dynamic RDMA Credentials;**

I. Introduction

On Cray XC® systems [1], interconnect network access is not easily shared between applications running in different job reservations. ALPS provides a *protection domain*(*pdomain*) feature that allows applications running with job reservations to share network access within two different contexts: system-wide or user. A protection domain is a network-level security mechanism used to prevent unauthorized access to network-mapped memory regions on a node. A system-wide *pdomain* is accessible by any application in any job reservation, which is inherently insecure. A user *pdomain* is accessible by any application run by a specific user in any job reservation, which is more secure but less flexible than a system *pdomain*. Both types of *pdomain* do not provide flexible sharing of network resources that also provide security to the applications using them. Additionally, Cray XC® systems may use SLURM [2] instead of ALPS. The *pdomain* feature is not present on SLURM, so sharing of network resources between different job reservations is not possible. To provide a secure and flexible solution to the problem of sharing network resources, we propose a system for sharing network resources called *Dynamic RDMA Credentials* (DRC).

Dynamic RDMA Credentials is a new system service to allow shared network access between different user applications using a managed network credential. DRC allows user applications to request managed network credentials, which can be shared with other users, groups or jobs. The credential provided by DRC is an object created by applications and managed by DRC to provide security by routing access requests to a credential through the DRC system. DRC consists of three new system services (`DRCC`, `DRCS`, and `DRCJEDI`), a library (`libdrc`), and a command-line utility (`DRCCLI`). `Libdrc` is used by user applications for run-time control of managed network credentials. User applications use library API functions to acquire, modify or release credentials. Credentials can also be managed outside of the run-time context using the DRC command line utility, `DRCCLI`[1].

Access to a credential is governed by the application and DRC to provide authorized and protected sharing of network access between applications. The `libdrc` API provides functions for acquiring new credentials and modifying access permissions for existing credentials. DRC maintains a list of authorized users, groups or jobs for each credential to prevent unauthorized access. User applications can grant access to other users, groups or jobs using the API during run-time, or through `DRCCLI`.

DRC extends the existing ALPS protection domain functionality without exposing application data to unauthorized applications. DRC solves existing privacy problems with *protection domains* by only allowing authorized users, groups or jobs to use a DRC managed network credential, so that only authorized applications may interact using the shared credential. Of further benefit, DRC is not dependent on ALPS and therefore can be used with other workload managers (WLM) such as SLURM. DRC extends the existing features found on systems using ALPS or SLURM to provide the same authorized and protected network access between applications.

In the following sections, we will show how DRC works, how to use DRC and demonstrate various examples of how DRC can be used. Examples will be given to show how users might interact with DRC to acquire, modify or release a managed network credential. More examples will be given to demonstrate the administrative capabilities of `DRCCLI`. Lastly, future work for DRC will be discussed including optimization work for scaling and performance, upcoming administrative features, and authorization enhancements.

[1] `DRCCLI` has a broader functionality than `libdrc` as it is intended to be a utility program for administrators. However, `DRCCLI` does not have all of the functionality of `libdrc` either.

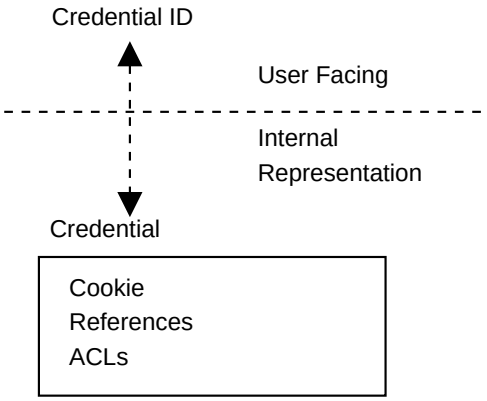

Figure 1. Representation of a DRC credential

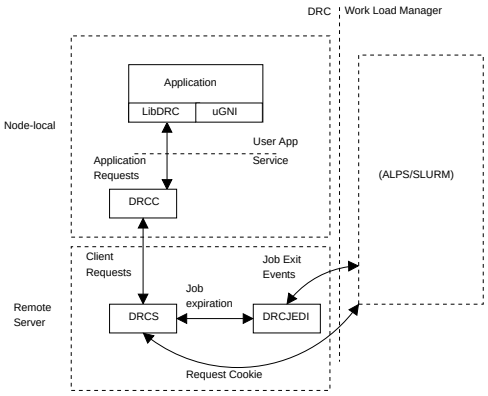

Figure 2. DRC system architecture as seen from a top-down perspective

II. Background

To explain how DRC functions as a system, it is important to understand how applications behave with and without DRC. In the next sections, we will discuss how a simple uGNI application would behave without DRC, then briefly cover the architecture of the DRC system, and finally, discuss how the same application would behave using the DRC system.

A. Simple uGNI application without DRC

A simple application that utilizes the `uGNI` library to communicate with the high-speed network (HSN) has many steps to perform before a message is ever sent. The first step in this process occurs outside of the application at the time when the job containing the application is scheduled with the work load manager (WLM). When the job is started by the WLM, the WLM supplies a private *protection tag* (*ptag*), and *cookie* for the job[2]. Once the application is started from within the job, the application can communicate with the WLM to retrieve the *ptag* and *cookie*. Once *ptag* and *cookie* is retrieved, the application uses the *ptag cookie* to create a `uGNI` *communication domain* (*CDM*). Once the CDM is created, any underlying objects created with the CDM are associated with *ptag* and *cookie* that the CDM was created with. At this point, the application creates the underlying objects that are needed to communicate across the HSN, and proceeds normally[3].

B. DRC architecture

To understand the changes made to the process that a simple application uses to access the HSN, we first need to discuss what DRC introduces to the system. DRC introduces multiple components to a system, as described in section I, that work together to provide secure, managed credentials. The credential provided by DRC is provided to the user as a `uint32_t` identifier that maps internally to a credential object[4] in the DRC system, as shown in figure 1. The DRC system, shown in figure 2, is composed of a client daemon (`DRCC`), server daemon (`DRCS`), job expiration detection daemon (`DRCJEDI`), and a library (`libdrc`).

The library, `libdrc`, is responsible for handling requests from the user applications. `Libdrc` is the portion of the DRC system that must be linked with the user application. Using `libdrc`, user applications may make requests for new credentials, modify permissions for existing credentials, or release resources[5]. `Libdrc` allows user applications to execute the following actions:

- acquire a credential (*acquire*)
- grant permission to a credential (*grant*)
- revoke access to a credential (*revoke*)
- access a credential (*access*)
- release credential reference (*release*)

These requests are processed wholly within the library, then forwarded as-needed to the DRC client daemon, `DRCC`.

`DRCC` services requests from `libdrc` for every application running on a node. When receiving the request, `DRCC` authenticates the user request before any processing occurs, preventing user applications from masquerading as other users, groups, or jobs. After authentication, `DRCC` will service any part of the request that can be processed locally. Some requests can be wholly satisfied within `DRCC`, preventing unnecessary network traffic and allowing the response to the application to be returned immediately.

[2]At the time that the job is scheduled, a user can supply a protection domain ID that changes the supplied *ptag* and *cookie* to the *ptag* and *cookie* associated with the protection domain.

[3]By 'normally', we mean that the application continues execution as defined by the program itself. Applications may choose to interact with the HSN as they choose

[4]In the case of tokenization(See section III-F), the credential can also be represented by a character string that contains encoded credential information.

[5]`Libdrc` provides two different functions for releasing resources: `drc_release_local` and `drc_release`. The former releases references to the credential and local *uGNI* resources, while the latter only releases the local *uGNI* resources.

If DRCC cannot fully satisfy the request, the request is forwarded to the DRC server daemon, DRCS.

DRCS services requests from DRCC and the DRC command-line interface, DRCCLI. DRCS is responsible for processing all requests for new credentials, modifications to existing credentials, or the release of system-wide resources[6]. To create a new credential, DRCS must communicate with the WLM to provision a new *cookie(s)*[7] for the credential. Releasing a credential follows a similar process where DRCS must inform the WLM that the *cookie(s)* are no longer needed and can be freed to a pool of free resources. Additionally, DRCS must communicate with the job expiration director, DRCJEDI, in the event that an application exits without calling the appropriate functions in libdrc.

The primary function of DRCJEDI is to inform DRCS of job exit events from the WLM. In the event that an application exits unexpectedly, the application may not inform libdrc (and by proxy, DRCS) that it has finished with its credentials. To ensure that DRCS knows when an application within a job has exited, DRCJEDI monitors the WLM for the events and forwards event notifications to DRCS. The job exit notifications allow DRCS to reconcile credential references and properly return credentials to the pool of free credentials when they are no longer in use.

C. Simple uGNI application with DRC

The same application running without DRC changes very little when running with DRC. An application compiled with libdrc can still use the WLM-provided *ptag* and *cookie*. However, the application can now request new credentials or access credentials that it has been granted access. The application first requests a new credential or attempts to retrieve a credential ID from another application through an out-of-band channel for which the application has requested and been granted access. Next, the application attempts to access the credential through libdrc using the *access* function. If the access is successful, the application can then use the libdrc helper functions to access the *cookie* information contained in the drc_info_handle_t returned from the *access* function. Once the *cookie* associated with the credential have been retrieved, the application can create a *CDM* that can be used to register *uGNI* communication elements necessary for inter-application communication[8]. Once finished with the uGNI communication elements associated with a credential, the credential should be released via the libdrc *release* function.

[6]System-wide resources from the perspective of DRC are protection domains, DRC credentials, *ptags*, and *cookies*.

[7]ALPS-based systems will provide two cookies, while SLURM systems only provide a single cookie

[8]The steps of creating and/or accessing a credential, then setting up the necessary *uGNI* communication elements can be done repeatedly for creating multiple shared communication domains.

III. Features

The basic functions of DRC need to supplemented to provide a robust and high-performance solution. To enhance DRC, the following features were implemented:

- Job expiration detection
- Administrative credential limits
- Persistent credentials
- Service restart reconciliation
- Node-local credential caching
- Credential tokenization
- Node-insecure mode

Job expiration detection is implemented as a resource monitoring feature, while other features such as credential tokenization are directly related to scalability and performance. In this section, we will discuss the various features available in DRC.

A. Job Expiration Detection

Typically, when an application exits, it should call the *release* API function in libdrc. However, in the event of unexpected application termination, the application may not make the appropriate calls to libdrc to manage the credential properly. Without releasing the application reference to the credential, it may remain allocated and holding network resources. To address this concern, DRC uses a feature called *job expiration detection*. Job expiration detection is a DRC feature that monitors the WLM for termination of job reservation and notifies the DRC server, DRCS, of these events.

DRC has a two-step approach to job expiration detection. The first step is to detect which applications are running within a job reservation, and using reference counting, associate them with credentials as necessary by taking a reference. References to a credential are taken when necessary during application calls to the *acquire* and *access* API functions. If an application is running from a job context, then DRC determines that the association with the credential is *transient*. Since jobs are limited in duration, the association of the application with the credential is also limited in duration, or transient. Given that the association is transient, DRC must take additional steps to ensure references to the credential are dropped when the associations are no longer present. Otherwise, no further action is necessary.

The second step is to detect and remove references of application associations with credentials when the job reservation expires. Irrespective of the WLM, DRC records the job reservation ID associated with an application when the application associates itself with a credential. The association becomes invalid after the application uses the *release* API function. Since DRC cannot detect abnormal termination, DRC instead coordinates with the WLM to determine what jobs have expired. Using the job reservation ID recorded earlier, DRC can remove the expired references

and determine whether the credential and its resources can be deallocated.

This feature is fully implemented within the DRC job expiration director service, `DRCJEDI`. The `DRCJEDI` service is fully compatible with ALPS and SLURM, and may be easily extended to other WLMs as needed.

B. Administrative Credentials Limits

Since DRC extends the existing *pdomain* functionality, credentials are limited by the same constraints. To address potential abuse by users or groups[9], DRC imposes administrator-controlled limits that are highly configurable.

Four different types of global limits exist: per-user, per-group, per-job and a global credential limit. A 'global credential' limit is a limit on the number of allocated credentials across the entire system. This allows administrators to effectively limit the portion of protection domains that DRC may consume out of the global pool. A 'per-user' limit is applied to each user on the system, regardless of any specific user limit. A 'per-group' limit is applied to each group on the system, regardless of any specific group limits. Lastly, a 'per-job' limit is applied to each job on the system, regardless of any specific job limits.

Outside of global limits, specific limits can also be set. Specific limits are user, group, or job limits. A 'user' limit is specific to the provided *UID*, a 'group' limit is specific to the provided *GID*, and a job reservation limit is specific to the provided job reservation ID.

All limits are independently verified. This means that if any one limit has been exceeded for a given *UID*/*GID*/*WLM_ID* tuple, a request for a new credential will be denied. This will allow administrators to easily set limits across a number of different parameters to encourage fair use of resources.

C. Persistent Credentials

By default, DRC credentials are created as temporary objects, existing only as long as there are outstanding references to the credential via *access*. If no references to the credential exist, then the credential is cleaned up and all allocated resources are freed. However, if the `DRC_FLAGS_PERSISTENT` is provided as a flag to the *acquire* call, then the credential will exist until *release* is explicitly called from the acquiring job or process. The credential will not go away immediately. Instead, the credential will exist until there are no remaining references to the credential, behaving in the same manner as normal credential. This is an optimization for applications that intend to share the credential over extended periods of time without requiring strict reference counting. Persistent credentials can survive reboots, and are functionally equivalent to system protection domains with access control.

[9]On systems utilizing ALPS, administrators may set a limit to the number of protection domains that may be allocated globally. This global limit may interfere with fair-use policies regarding network resources in some cases.

D. Service Restart Reconciliation

As part of the job expiration detection feature, `DRCJEDI` notifies `DRCS` when a job reservation has terminated so that `DRCS` may take necessary action for resource management. However, if the `DRCJEDI` service has stopped or is in the process of restarting, notifications from `DRCJEDI` to `DRCS` may be lost. Since credential information must persist between restarts of the service, `DRCS` uses a persistent database to store credential information and lost notifications may cause the persistent store to become out-of-sync with the WLM job information. In order to ensure that resource management of credentials is done properly, `DRCJEDI` will attempt to reconcile the persistent credential information with `DRCS` by actively requesting job reservation information for existing credentials.

The process for reconciliation is simple. In the event of a process restart or node reboot, `DRCJEDI` sends a request to `DRCS` for a list of job reservation IDs that are associated with existing credentials. Next, `DRCJEDI` requests the list of active job reservations from the WLM, then compares the list from the WLM with the list of "outstanding" job reservation IDs from `DRCS`. A list of expired job reservation IDs is built from the set of job container IDs that do not exist in the active list from the WLM but exist in the list of "outstanding" job reservation IDs from `DRCS`. Finally, the list of expired job reservation IDs are sent to `DRCS` to reconcile the database records. The end result of the reconciliation is the re-synchronization of WLM information with the `DRCS` database information about job reservations. The reconciliation allows `DRCS` to release network resources held by a DRC credential if the credential was transient and no references to the credential exist.

To support the persistent credentials feature, service restart reconciliation has been expanded to cover `DRCC` as well. In the event that the `DRCC` service is stopped or restarted, `DRCC` will attempt to reconcile any persistent credential references with the `DRCS` server. In the event that the node running the `DRCC` service has rebooted, `DRCC` will notify `DRCS` that the node was rebooted and any references held by the node should be dropped. In the event that only the `DRCC` service was restarted, `DRCC` can check the process list against an internal credential cache to verify which applications have exited. This type of operation has much less impact to `DRCS` and may not require any notification at all. In either case, `DRCS` is notified of changes to the references so that network resources can be released if no longer needed.

E. Node-local Credential Caching

A naïve approach to handling *access* requests would require each instance of an application using `libdrc` to coordinate with the DRC server, `DRCS`. Unfortunately, this approach is not scalable, especially for small jobs where the number processes per node is high.

To provide scalability on a per-node basis, DRC utilizes a cache of credential information within DRCC to prevent spurious communication with DRCS. When an application makes an *access* request via the API, it first checks the credential cache within DRCC. If the inbound credential *access* request has been sent before and the credential association hasn't been released, then DRCC does not need to forward the *access* request. This optimization reduces the number of requests to the server by the number of processes-per-node for a given job. This reduces the overall request processing done by DRCS and allows DRC to scale better on larger systems.

F. Credential Tokenization

Node-local credential caching is of little benefit for the traditional HPC use case in which jobs span large numbers of nodes with one or very few processes per node. For this scenario, further involvement is required from the client software. To address this case and further improve scalability, DRC provides credential tokenization.

Credential tokenization is a method of encoding DRC-specific credential information into a buffer and returning it to the client. The client can then distribute the token to other processes within the same job. Once another process has the token, it can use a variant of the *access* API with the token. The variant of the *access* API function accepts credential tokens and does not require coordination with DRCS, eliminating traffic on the network and further reducing the amount of request processing done.

G. Node-insecure mode

Node-insecure mode is a feature that allows applications from multiple jobs on the same node to use the same credential. DRC credentials are created with the perspective of one job reservation or application set per node, which allows DRC to tightly control access to the credential. By allowing only a single job reservation ID per cookie, DRC can prevent unauthorized jobs from attempting to bypass DRC access control. This type of security violation could could occur on systems using multiple-user,multiple-job(MAMU) capable WLMs if the cookie is not bound to a job reservation. If the cookie were passed to an application in a different job reservation on the same node, the application could directly configure uGNI using the cookie and access the HSN, even if they had not been granted access via DRC. As a flag to *acquire*, DRC provides the ability to change the default behavior and provide the 'Node-insecure mode' feature.

Node-insecure mode creates a credential that is not bound to a specific job reservation. Since the credential is not bound to a job container, any job may configure using the cookies returned in the credential. Any job may do this, and thus it is unnecessary to use DRC further. This feature is provided as an exception to the process for use cases which require multiple applications on the same node to use the same credential. Users of this feature are recommended to use caution to when passing credential information for credentials allocated with node-insecure mode and to never pass cookie information in an unsecured manner. DRC cannot control cookie-level information and as such cannot protect against unauthorized access to the network if security measures are bypassed when using node-insecure mode. Access control through DRC is still possible if cookies are only retrieved using the DRC *access* API function.

IV. Usage

In this section, we will discuss how to interact with the DRC system, provide an overview of the API and command-line interface, and provide examples of simple uses of DRC.

A. Functions

DRC provides a set of functions for manipulating the managed credentials that may be used from the API or command-line interface. This section will briefly define the behavior of each action and how it interacts with the DRC system. At the time of this paper, the following core functions have been provided: *ACCESS*, *ACQUIRE*, *GRANT*, *LIMIT*, *LIMIT*, *REVOKE*, and *RELEASE*. The *ACCESS* function is specific to libdrc and is not available from the command-line interface, DRCCLI. The *LIST* and *LIMIT* functions are specific to DRCCLI and are not available from libdrc.

ACQUIRE requests a managed credential from DRC. Credentials may be acquired up to the credential limit for the calling user. If the credential limits for the calling user have not been exceeded and available credentials exist, a credential identifier will be returned. See section III-B for more details on limits.

GRANT modifies the credential access control lists for a specific credential ID to add a new user, group, or job reservation to the list of entities which may access the credential.

REVOKE modifies the credential access control lists for a specific credential ID to remove a user, group, or job reservation from the list of entities which may access the credential. This does not terminate access for existing references to a credential, but prevents new access requests from the denied entity.

RELEASE removes a reference held by the calling application for a specific credential ID. Once all references to a credential have been dropped, any network resources held by a credential will be released to the underlying provider. Libdrc provides a 'local' *RELEASE* in addition to the standard *RELEASE*. The libdrc local release allows applications to return node-local network resources in the event that the credential is still needed but the node-local resources are not.

ACCESS requests access to a specific credential ID. Access to a credential is accepted if the user, group, or job reservation has been granted permission to the credential. The user or job reservation that acquired the credential via *ACQUIRE* is implicitly granted access to the credential. On success, *ACCESS* configures the node for HSN access and provides information to the application needed to configure uGNI access for the ARIES NIC.

LIST requests information about DRC credentials. The information provided by this function can be used by administrators for diagnostic purposes. This function is provided as a way to get an overview of the current state of a DRC system.

LIMIT allows administrators to configure credential limits in a DRC system. See section III-B for more details on limits.

B. Application Programming Interface

DRC provides a C library API via `libdrc`. `Libdrc` provides the ability to *ACQUIRE*, *GRANT*, *REVOKE*, *ACCESS*, *RELEASE* to applications that are linked with `libdrc`. `Libdrc` also provides a set of helper functions that allow applications to retrieve additional information about credentials. By using `libdrc`, applications can create new credentials, grant/revoke access to existing credentials, release local resources or credential references, and access existing credentials. After accessing an existing credential, applications can retrieve the shared *cookie* from the credential and use the *cookie* to configure HSN access. This process allows for easy setup of a shared network domain between applications running in different contexts.

To improve user experience, the API is stable between point releases and the library can be updated without requiring applications to recompile between updates[10].

C. Command-line Interface

A command line utility, `DRCCLI`, is provided for manipulation of the DRC system outside of the context of the C library. `DRCCLI` was created to allow for administrators to control `DRCS`. Using `DRCCLI` an administrator can use the following verbs: *ACQUIRE*, *RELEASE*, *GRANT*, *REVOKE*, *LIST*, and *LIMIT*.

D. Examples

In this section, we cover different examples of how to use the API from different perspectives and use cases, such as tokenization. Additionally, we show an example use case for `DRCCLI` to achieve some of the same behavior performed in the API examples, and demonstrate ways that `DRCCLI` can be utilized by the administrator. As these examples are primarily for demonstrating how a user might interact with DRC, examples will not show proper error handling for the sake of brevity.

[10]Major API changes may require recompilation if function definitions for existing functions are modified. Where possible, existing functions will be preserved and alternative functions will be provided.

```c
#include "rdmacred.h"

int main(int argv, char** argv) {
  drc_infohandle_t cred_info;
  uint32_t credential, cookie1, cookie2;
  uint8_t ptag1, ptag2;
        ptag1 = ptag2 = GNI_FIND_ALLOC_PTAG;

  drc_acquire(&credential, 0);
  drc_access(credential, 0, &cred_info);
  cookie1 = drc_get_first_cookie(cred_info);
  cookie2 = drc_get_second_cookie(cred_info);

  GNI_GetPtag(0, cookie1, &ptag1);
  GNI_GetPtag(0, cookie2, &ptag2);

  // Configure Normal UGNI Application here
  // Grant access to client applications using
    following
  drc_grant(credential, inc_data.cred_wlmid,
    DRC_FLAGS_TARGET_WLM);
  // Do Application Work
  drc_release(credential);
}
```

Figure 3. Example Acquiring Program

```c
#include "rdmacred.h"

int main() {
  drc_info_handle_t info;
  uint32_t credential, cookie1, cookie2;
  uint8_t ptag1, ptag2;

  // Acquire credential id and permission through
    OOB service
  rc = drc_access(credential, 0, &info);
  cookie1 = drc_get_first_cookie(info);
  cookie2 = drc_get_second_cookie(info);

  grc = GNI_GetPtag(0, cookie1, &ptag1);
  grc = GNI_GetPtag(0, cookie2, &ptag2);
  // Configure uGNI and proceed normally
}
```

Figure 4. Example basic client

The first example, shown in figure 3, demonstrates how an application might acquire a credential, then grant access to applications in another job reservation. The first step to allocating a DRC credential is the call to `drc_acquire`. Then, the application calls `drc_access` to access the credential. At this point, the application can use the information passed back in `cred_info` and the helper functions provided by `libdrc` to extract information necessary to configure HSN access using uGNI within the application itself. Next, the application grants access to any application running in a different job reservation using the job reservation ID given

```c
#include "rdmacred.h"

int main(int argc, char **argv) {
  uint32_t credential;
  drc_info_handle_t info;
  char *token;

  if (server) {
    // Acquire credential id and permission
    through OOB service
    drc_access(credential, 0, &info);
    drc_get_credential_token(credential, &token);
    // Send token through OOB to other ranks
  } else {
    // Get Token through OOB source
    drc_access_with_token(token, 0, &info);
  }
  // Do normal application work here ... then
    release at end
  drc_release(credential, 0);
}
```

Figure 5. Example client scaling using tokenization

```
user@machine> module load rdma-credentials
user@machine> credential=$(drccli acquire)
user@machine> drccli grant -u $(id -u otheruser)
    $credential
user@machine> drccli list
Credential Sys_Id     uid       gid   wlmid
    Cookie1      Cookie2    State     Refs
          1 drcs_1        0         0       0    0
    x2c20000    0x2c30000    READY        1
user@machine> drccli release 1
user@machine> drccli list -c $credential
Credential Not Found
```

Figure 6. Example DRCCLI usage

by `inc_data.cred_wlmid`[11] in the example, before proceeding on to other work that application might need to do. Finally, as the application closes, the reference to the credential should be released with `drc_release`.

The second example, shown in figure 4, demonstrates how client application would access credentials that have already been acquired but not shared. As only one application in a job needs to acquire a credential, this example shows how other ranks of a job might behave when attempting to access the credential. In this example, the client application has knowledge of an external server, service or application that holds information about a DRC credential that the client application wishes to use. In an out-of band channel, the client application requests permission for the credential and retrieves a DRC credential ID. Next, the client application uses the received credential ID and `drc_access` to attempt to access the credential through DRC. If successful, the client application can use the `libdrc` helper functions and the `info` handle to retrieve the cookie information necessary to configure the application's HSN access with the uGNI library. Once uGNI is configured with the information provided from the credential, the application proceeds normally. As mentioned in section III-A, client applications should release the credential before exit.

As part of the scaling optimizations, applications utilizing DRC may opt to use the *credential tokenization* feature described in section III-F.The next example, shown in figure 5, demonstrates how an application might use the *credential tokenization* feature. Similar to the previous example, this example shows how an application would access credentials that have already been acquired and not shared, but with the additional caveat that this example will use tokenization to achieve the same end goal. Starting in the same manner, the client application must request permission to use the credential from the credential owner and retrieve the credential ID through the out-of-band channel. With tokenization, only one rank of a job needs to call `drc_access`, a function that requires some communication to the `DRCS` service. So in this example, one rank serves as a 'server' to the rest of the ranks by calling `drc_access`, then proceeds to get the credential token by calling `drc_get_credential_token`. The token acquired by the 'server' rank must then be communicated to the other ranks of the job through an out-of-band channel. Switching to the 'client' context, clients must wait for the credential token to arrive from the out-of-band channel. Once the credential token has been received, 'client' ranks can call `drc_access_with_token` in the same manner that `drc_access` was used in prior examples, but without the extra communication to the `DRCS` service. Finally, both the 'client' and 'server' ranks can proceed to use the information provided by `info` and the `libdrc` helper functions to configure HSN access with uGNI and proceed normally[12]. Similarly to all other examples, the client application should call `drc_release` before exit.

The last example, shown in figure 6, demonstrates how an administrator might use the `DRCCLI` utility[13]. In this example, the `rdma-credentials` module is loaded prior to executing any commands[14]. Next, a credential is requested using the `DRCCLI` and the *acquire* verb, and the credential ID is printed and stored in the 'credential' variable. Next, the access to the new credential is granted to a user via the *grant* verb, and then the contents of the DRC credential list is displayed using the *list* verb. Finally, the credential is released via the *release* verb, and the *list* verb is used to

[11]The variable in the example is not one that is provided by DRC. The WLM ID to be granted access, as well as any other identifiers, should be communicated via an out-of-band channel or known in advance.

[12]Once the *access* function calls are complete, application behavior between ranks be identical with respect to the DRC system. DRC imposes no distinction between ranks of a job.

[13]As of the initial release, `DRCCLI` is limited to users with `root` access. This limitation may be addressed in a future release depending on feedback

[14]This step is only required where the module is not loaded by default. Administrators may choose to add the module load step where appropriate, and may choose to include the step in the default profile initialization.

confirm that the credential was fully released[15].

V. Conclusion

In this paper, we have presented the implementation of the DRC system and component parts of the system. DRC provides a secure solution for sharing network access on ALPS and SLURM-based systems that was not previously available on Cray XC systems. On ALPS-based Cray systems, DRC enhances existing capabilities to provide secure shared credentials using the ALPS protection domain feature. Additionally, DRC provides secure shared credentials for SLURM based systems, which was not previously available on Cray XC systems. As part of the first release, many features will be introduced that provide users with a scalable, highly configurable, and secure solution for managing the shared network credentials that DRC offers. Features such as *job expiration detection* ensure that network resources are managed with proper accounting to prevent unintentional resources leaks and security violations. Examples have been given to demonstrate the various ways that DRC can be used from a user's perspective. Sample code snippets and script executions have been provided to show potential uses of `libdrc` and `DRCCLI`.

VI. Future Work

As the DRC system is further developed, we will provide additional features and enhancements to promote scaling, performance, and functionality. Specific features might include multi-threading the `DRCC` service, fail-over for `DRCS`, and multi-node/multi-process `DRCS`. Enhancements such load-distribution for the `DRCS` service may serve to increase performance and promote scaling goals. Expanding the capabilities of the command-line interface is another feature that might serve to increase the functionality of the DRC system. It would involve allowing non-administrative users to allocate credentials outside of the application context, and allow them to administer those credentials.

Currently, persistent credentials can only be allocated by the administrator or system services. We would like to expand this feature to users or groups that an administrator specifically permits, allowing administrators to provide more control of DRC to the users while still maintaining full control over the system.

We will also take user feedback into account once the service is delivered to customers, as we would like to make it as useful as we can.

Acknowledgment

The authors would like to thank Cray Inc. for providing an excellent place to work and create new technologies. We would also like to thank our families for their love and support.

[15]Credentials may still be present in the list if an application owned a reference to the credential and was still using the credential. Credentials will be deallocated automatically once the last reference to the credential has been released.

References

[1] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC Series Network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[2] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.