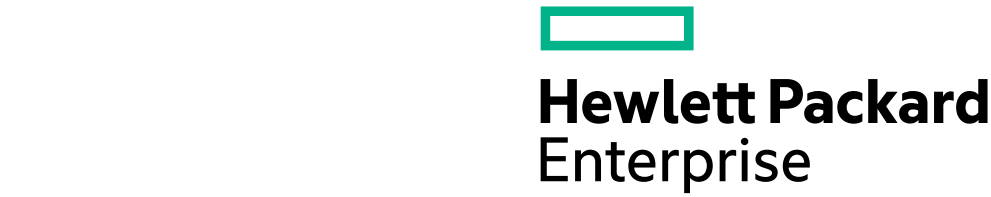
Software Architecture For The Machine

Version 2.1, 2016-04-21



Keith Packard

Software Architecture For The Machine: Version 2.1, 2016-04-21

Keith Packard

Revision History		
Revision 2.0	29 January 2016	KeithPackard
Add management services sections.		
Revision 1.0	22 October 2015	KeithPackard
Use fam-atomic library documentation from source. Publish versioned URLs		
Revision 0.7	13 October 2015	KeithPackard
Respond to review comments, clarifying POSIX interactions and LFS limitations.		
Revision 0.6	11 October 2015	KeithPackard
Add librarian control API Build RVMA docs from rvma sources		
Revision 0.5	10 September 2015	KeithPackard
Add spinlocks to fam-atomic library		
Revision 0.4	14 August 2015	KeithPackard
Add POSIX API changes Define fam-atomic library Add pmem		
Revision 0.3	8 August 2015	KeithPackard
Add atomics, flushing and RVMA		
Revision 0.2	16 June 2015	KeithPackard
First tagged version.		

Table of Contents

List of Figures

List of Tables

Chapter 1. This Document

This document is written in AsciiDoc and maintained in the hlinux GitLab instance at

https://hlinux-gitlab.us.rdlabs.hpecorp.net/l4tm/software-arch

1.1. Making Changes

Here are the steps required to make changes to this document:

1. Proposed changes shall be submitted as git patches relative to the above repository to the l4tm arch council email list: latc@hp.com [mailto:latc@hp.com]

2. Proposed changes shall be reviewed by members of the l4tm arch council with comments sent back to the author and the list.

3. Proposed changes ready for inclusion shall be marked as *Reviewed-by:* a member of the l4tm arch council on the list, using the standard git conventions.

4. All changes marked as *Reviewed-by:* and not yet incorporated into the spec shall be discussed at the l4tm meeting.

5. Changes which receive a rough consensus of the l4tm arch council in favor of inclusion shall be merged into the repository.

6. Changes which receive a rough consensus of the l4tm arch council against inclusion shall be discarded.

7. Changes which do not reach rough consensus will be sent back to the author for further revision along with a summary of the comments collected during the meeting.

1.2. Formatting this document

Building the pdf and html versions of the document require a small set of asciidoc and docbook tools, along with graphviz to generate some of the graphics.

1.3. Stable download locations

The most recent ERS version is published at:

http://hlinux-build.us.rdlabs.hpecorp.net/~packardk/software-arch.pdf

and

http://hlinux-build.us.rdlabs.hpecorp.net/~packardk/software-arch.html

This specific version is published at:

http://hlinux-build.us.rdlabs.hpecorp.net/~packardk/software-arch-2.1.pdf

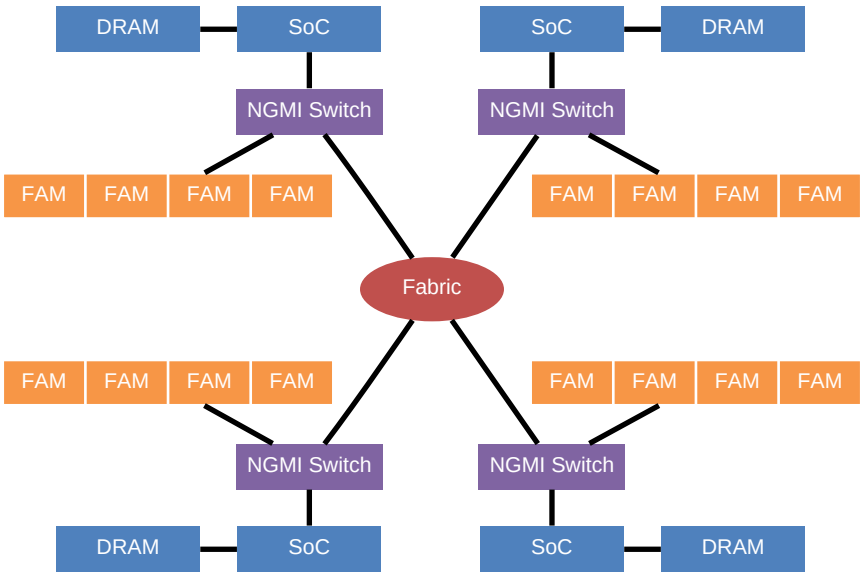
and

http://hlinux-build.us.rdlabs.hpecorp.net/~packardk/software-arch-2.1.html

Chapter 2. The Machine

For purposes of this document, the following simplified hardware architecture will suffice:

Figure 2.1. Hardware Architecture

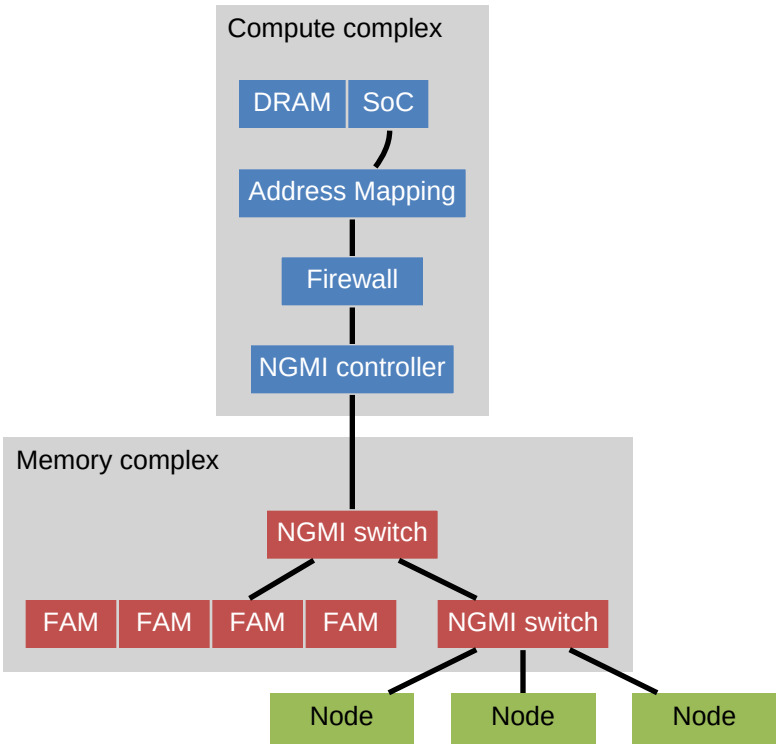


At the heart of the system lies the Fabric, which is constructed by the interconnection of many nodes through GenZswitch devices.

Not shown in this diagram are various management infrastructure bits within the overall initial system implementation. These include a COTS top-of-rack management server, networking connections between the nodes and with the top-of-rack management server, management processors on each node and the separate management network.

In the current implementation of The Machine, each node within the machine is split into two pieces; compute (SoC plus DRAM) and storage (Fabric interconnect and Fabric Attached Memory (FAM)). The following diagram shows how the pieces within the node connect.

Figure 2.2. Node Components



The compute complex includes an address mapping unit which serves to take physical addresses from the SoC and map them into global Logical Z addresses (LZA). This serves to both allow for a larger LZA space than the SoC can directly address as well as providing a mechanism for mapping discontiguous regions of FAM into contiguous physical address (PA) ranges within the SoC. The address mapping unit is controlled by the node operating system. Address mapping may be done in 8 GB Book and 64 kB Booklet units.

Access to Fabric Attached Memory (FAM) is controlled by the Firewall, which specifies which books the SoC is permitted to access. It is important to note that even access to the FAM located within the same node is controlled through the Firewall.

A collection of nodes which share access and addressing to the same set of FAM is called a Load/Store Domain (LSD).

Chapter 3. Licensing and Distribution Policies

System Software for The Machine will be implemented using an open architecture. This means that all software providing essential system services will either be free software or implement an publicly documented API.

3.1. Licensing

Changes to existing free software projects will be released under a license compatible with the related project or projects. Upgrading to a compatible share-alike license is encouraged. Changes to existing free software projects under non share-alike licenses should be released. Exceptions to this policy will need approval.

New free software projects should use a share-alike license, either GPL, LGPL or GPL with class-path exception as appropriate. Exceptions to this license policy will need approval.

Non-free software will need approval. Interfaces to this software will need comprehensive documentation and review to verify that the interfaces are fully documented to the point where an inter-operable implementation could be constructed solely from the published interface specifications.

3.2. Distribution

Changes to existing software projects should be merged upstream. This will likely require close cooperation and collaboration with the relevant upstream maintainers to get changes prepared in a fashion that can be merged.

New software projects should be made available in a suitable external environment with documentation and source code made available for download. Regular updates to the software should be published and announced on suitable public discussion lists or web sites.

Each free software project should plan to migrate to a public issue tracker for all software issues. For existing projects, issues should be tracked in an existing system where practical.

3.3. Development

Work on free software projects, both new and existing, should move to publicly visible source code repositories so that customers and partners can view the entire history of development. Exceptions to this policy will need approval.

A suitable public mailing list or on-line forum should be selected or created for each project. Internal conversations about the development of the software should plan on migrating to these external lists.

Internal projects that include changes to existing free software projects should produce a plan for getting changes readied and merged upstream as soon as practical.

3.4. Packaging

All Software documented in this specification shall be included in The Machine Distribution (TMD). TMD is a Debian-based distribution, which means that all software must be packaged in a *.deb* file, uploaded to the HPELinux archive and validated for inclusion in the archive.

Chapter 4. To Do List

These are items captured from various meetings for this document:

4.1. ERS 2.1 issues

These need to be addressed for version 2.1 of this document.

4.1.1. Secure Boot (Bill Hayes)

Document the TMD/L4TM SecureBoot plans

4.1.2. Memory Access Security

Pratyusa to review the TMD ERS 2.0 for the memory access control architecture to assure it meets Security Lab's requirements.

4.2. MFT′

These are issues to be resolved for work past the full-rack demo.

4.2.1. Performance counters.

Vulcan performance counters to go into to the Linux perf tools. 3 classes:

1. ARM-architected,

2. DRAM perf counters,

3. ICI perf counters

The latter 2 may not be publically available. But since those are not open source – we can't expose those APIs legally – and L4TM is all open source so we cannot work with that data. As spoken HPE Legal. So, ARM-architected are items we can use. Approach for closed data is to work with the vendors for them to expose or open-source those counters.

4.2.2. Discovery

Need a management service to collect all the FW configuration data with which to build the configuration file.

FW must provide a redfish API to read discoverable configuration information. ACTION: to Steve Lyle to specify this API – Drew Walton to assist.

Steve Lyle to enter the FW RESTful APIs into the MFT ERS.

Chapter 5. Configuration Data

The Machine configuration includes details including how much memory is installed on each node and how that memory is addressed, along with network addressing information in the form of MAC and IP addresses. It also includes services configuration information for accessing the various management services. Finally, there are a number of TLS keys involved in network communications among the software elements.

All of this data needs to be available to tasks running both on the management server as well as within each node.

5.1. Requirements

Store and distribute machine configuration information to processes running on the management server and within the nodes.

All hardware definitions include the associated global coordinate.

5.2. API

The Assembly Agent provides RESTful interfaces for fetching some of this information. See Section 6.6, “Assembly Agent” for details.

Python convenience interfaces for accessing this data will be defined and are expected to remain stable through future implementations.

5.3. Initial Implementation

Configuration data for the memory and services will be stored in a single JSON-formatted file, installed in a standard place on every operating system instance within the machine including manifests for nodes and the management server itself.

The current configuration file will be included in OS manifests instantiated on nodes. The configuration file contents must not be frozen when the manifest is defined, but must be dynamically incorporated as the manifest is delivered to the node for booting.

The configuration file will be stored in /etc/tm-config and be readable by all users and writable only by root.

5.4. Contents

While delivered in a single file, the configuration database is logically split into three major sections:

1. Hardware Descriptions

2. Memory Configuration

3. System Services

The structure of these will be described in outline form with nested contents (described later) elided with ellipsis.

The top level structure will define the overall address of the machine instance, including a version number which will track changes and is split into two parts, a major version and a minor version. The major changes for backwards-incompatible changes, the minor version changes for backwards-compatible changes. We aspire to never change the major version.

This document describes version 1.0 of the configuration file.

```
{
        "_comment": "Coordinates are expressed in a way that requires concatenation with their parent
        "coordinate": "machine_rev/1/datacenter/pa1",
        "version": "1.0",
        "racks":
        [
                …
        ],
        "interleaveGroups":
        [
                …
        ],
        "managementServer":
        {
                …
        },
        "advancedPowerManager":
        {
                …
        }
}
```

5.4.1. Hardware

Rack data. The configuration file includes a complete description of all hardware collected for a particular machine instance as a list of *racks*:

```
"racks":
[
        {
                "coordinate": "frame/A1.above_floor/rack/1",
                "enclosures":
                [
                        …
                ]
        },
        …
],
```

Enclosure data. Each rack is a collection of *enclosures*:

```
"enclosures":
[
        {
                "coordinate": "enclosure/1",
                "iZoneBoards":
                [
                        …
                ],
                "nodes":
                [
                        …
                ],
        },
        …
],
```

Innovation zone boards: Each enclosure holds one or more *innovation zone* boards:

```
"iZoneBoards":
[
        {
                "coordinate": "izone/1/izone_board/1",
                "izBoardMp":
                {
                        "coordinate": "izboard_mp/1",
                        "ipv4Address": "10.254.1.101",
                        "mfwApiUri": "http://${ipv4Address}:8080/redfish/v1",
                        "_comment": "msCollector entries are used by the Monitoring Service to identi
                        "msCollector": "switchmp"
                }
        },
]
```

Nodes. Each enclosure holds a set of nodes.

```
"nodes":
[
        {
                "coordinate": "node/1",
                "serialNumber": "<serial number for node>"
                "soc":
                {
                        …
                },
                "nodeMp":
                {
                        …
                },
                "mediaControllers":
                [
                        …
                ]
        },
        …
],
```

SoC Configuration Data.

```
"soc":
{
        "coordinate": "soc_board/1/soc/1",
        "tlsPublicCertificate": "<Base64 encoded certificate>"
        "macAddress": "xx:xx:xx:xx:xx:xx"
},
```

Node management processors:

```
"nodeMp":
{
        "coordinate": "soc_board/1/node_mp/1",
        "ipv4Address": "10.254.1.201",
        "mfwApiUri": "http://${ipv4Address}:8080/redfish/v1",
        "msCollector": "nodemp"
},
```

Psylocke Configuration Data.

```
"mediaControllers":
[
        {
                "coordinate": "memory_board/1/media_controller/1",
                "memorySize": "32G"
        },
        …
]
```

5.4.2. Memory Configuration

This describes the collection of memory comprising each interleave group in the system and forms the lowest level location possible for locating memory allocation within the librarian. Interleave groups are configured by the firmware.

Because the interleave group configuration spans across physical groupings of hardware, location information for each psylocke within the group uses a complete global address, rather than the relative addresses used within the hardware hierarchy.

```
"interleaveGroups":
[
        {
                "groupId": 1,
                "mediaControllers":
                [
                        "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                enclosure/1/node/1/memory_board/1/media_controller/1",
                        "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                enclosure/1/node/1/memory_board/1/media_controller/2",
                        "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                enclosure/1/node/1/memory_board/1/media_controller/3",
                        "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                enclosure/1/node/1/memory_board/1/media_controller/4"
                ]
        },
        …
]
```

(The coordinates were split here to fit on the page.)

5.4.3. System Services

In keeping with the overall physical layout-based structure from the hardware, this description of the hardware providing system services is based on physical boxes with coordinates. This is a slight variation from the original scheme proposed by Rycharde as it introduces the possibility of the set of services being spread across multiple servers. There is a restriction, however, that each service must be present only once in the system.

Here's the array of servers:

```
"servers":
[
        {
                "_comment": "The top-of-rack management server",
                "coordinate": "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/racku/41"
                "ipv4Address": "10.254.1.1",
                "softwareRecipeUri": "file:softwareRecipe.js",
                "services":
                [
                        …
                ]
        },
        …
]
```

Each server supports an array of services:

```
"services"
[
        {
                "service": "keyManager",
                "_comment": "The Key manager has RESTful API",
                "restUri": "https://:12000/rest/v1",
                "tlsPublicCertificate": "<Base64 encoded certificate>"
        },
        {
                "service": "librarian",
                "_comment": "The librarian has a custom JSON API",
                "port": 5432,
                "bookSize": "8G",
                "tlsPublicCertificate": "<Base64 encoded certificate>"
        },
]
```

Table 5.1. Service Objects

Name	Contents
service	Name of the service
restUri	The base URI for a RESTful service
port	TCP port for a non-RESTful service

Name	Contents
protocol	Protocol for non-RESTful service
version	Version of the protocol
ipv4Address	Alternate IP address (use servers by default)
tlsPublicCertificate	Public certificate used to authenticate service on connection
bookSize	For librarian, the minimum allocation size
pauseBeforePower	**ask rycharde**

Table 5.2. Services

Service Name	Objects
keyManager	restURI, version, tlsPublicCertificate
identity	port, protocol, version, tlsPublicCertificate
authorization	restURI, version, tlsPublicCertificate
osProvisioning	restURI, version, tlsPublicCertificate
librarian	port, protocol, tlsPublicCertificate, bookSize
assemblyAgent	restURI, version, tlsPublicCertificate, pauseBeforePower,
monitoring	restURI, version, tlsPublicCertificate
powerManager	port, version, tlsPublicCertificate

5.5. Future Implementation

All access to the data will be made through RESTful interfaces to either the Assembly Agent or other services running on the management server.

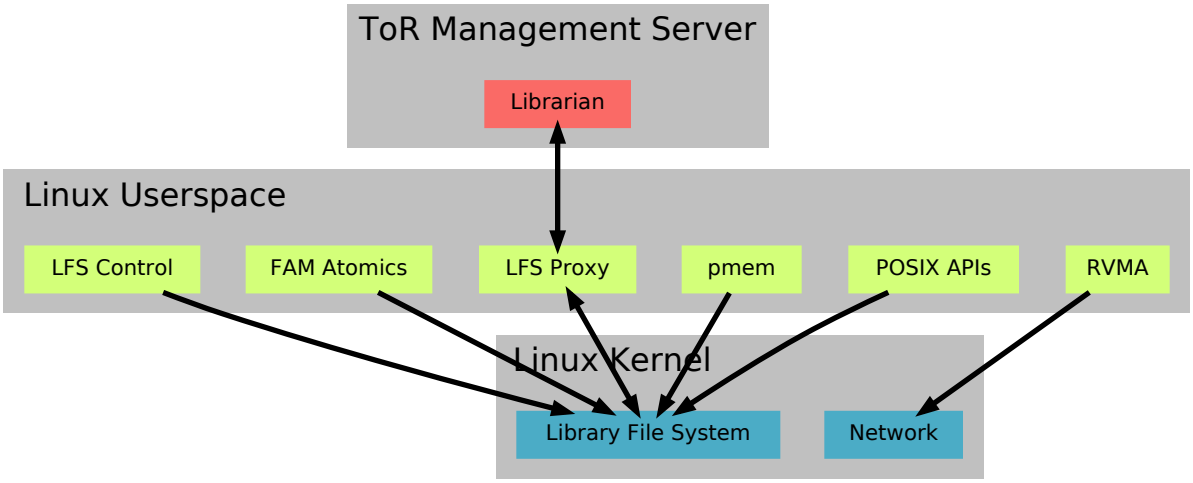
Data about the hardware environment will be automatically discovered at startup and dynamically updated while running.

Chapter 6. Global Services

To make a collection of nodes within the machine function as a unified computational platform, some central services are needed.

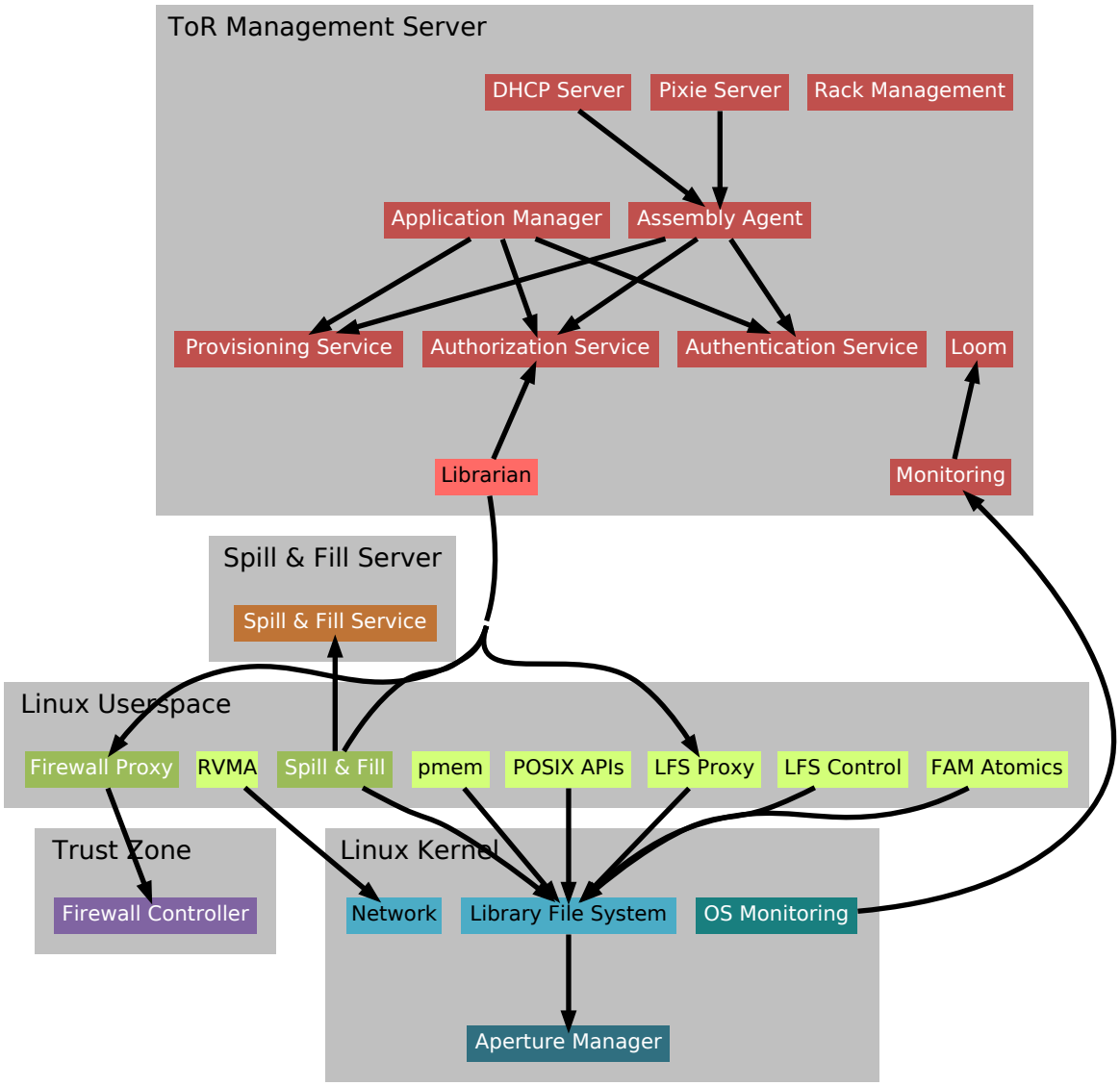
As of version 1 of this specification, the only service provided is the Librarian. A diagram of that is visible in Figure 6.1, “Software Architecture, Version 1”

Figure 6.1. Software Architecture, Version 1



For version 2, the remaining system service functionality will be incorporated, creating a system that looks like Figure 6.2, “Software Architecture, Version 2”

Figure 6.2. Software Architecture, Version 2



6.1. Authentication Service

The Authentication Service provides a global name for agents within the machine environment. An “agent” can be a user or role.

6.1.1. Requirements

Provide naming and authentication for agents within the machine environment. This includes:

1. Administrators. Agents with ssh access to the management server and administrative access to all of the management services.

2. Tenants. Agents who can manifest images and request node boot/shutdown.

3. Users. Accounts on nodes within the machine. These are included here strictly as a convenience for early deployments of the software.

Store authentication data for each of the above, allowing applications to verify credentials:

1. POSIX password.

2. SSH public keys.

As noted above, user accounts for nodes within the machine are going to be managed within the global authentication service during early development of the machine software systems. This enforces a global administrative domain over all user accounts, which is only viable while we have only a single tenant on the machine itself.

6.1.2. Dependencies

None

6.1.3. Hosting

For current hardware platforms, the top-of-rack management server will run the standard slapd daemon to provide LDAP services.

For future hardware platforms, we may choose to implement this as a federated service running across the whole Load/Store Domain (LSD) (or even wider). Within the LDAP world, there are many systems designed to provide precisely this kind of redundancy.

6.1.4. API

RFC 1823 specifies the network interface to an LDAP service.

For C (and C++), openldap provides a standard library, packaged in debian as libldap.

For Python, there are two separate ldap implementations, the older python-ldap, which works only with python 2 and the newer python-ldap3/python3-ldap3, which works with both versions. Python-ldap wraps the C ldap library while python-ldap3 is written in python directly.

Because we’re trying to use python3 everywhere, we’ll need to use python-ldap3.

For Java, there are at least three libraries already packaged for Debian:

1. libapache-directory-api-java Provided by Apache; seems to mostly work, but is released under the Apache 2.0 license only.
2. libspring-ldap-java Quite the dependency stack for this, it looks too heavyweight for most projects that aren't more tied to other libspring stuff
3. libvt-ldap-java Virginia Tech library, seems at about the same level as the apache one but is dual licensed Apache 2.0 or LGPL 3. This project has been renamed to *ldaptive* and moved to http://www.ldaptive.org/ but has not been packaged for Debian. The ldaptive API is significantly different than the libvt-ldap-java API though.

Because of licensing issues, we should avoid libapache-directory-api-java and instead plan on either sticking with the existing libvt-ldap-java module, or packaging ldaptive for Debian and using that instead.

Higher Level APIs

We may want to provide a simple layer on top of the basic LDAP interface that performs authentication given a username and password.

Applications interested in access control should consider whether the Authorization Service model is appropriate.

6.1.5. Additional Schema

The default *nis* schema provides for POSIX users, groups and passwords.

The version of SSH in Debian already has the hooks necessary to support LDAP through the AuthorizedKeysCommand configuration value. To support storing SSH keys in LDAP, we'll need to pick one of the many schema available in the wild:

- openssh-lpk (https://code.google.com/p/openssh-lpk/)
- openssh-ldap-publickey (https://github.com/AndriiGrytsenko/openssh-ldap-publickey)
- ssh-ldap-publickey (https://github.com/jirutka/ssh-ldap-pubkey)

The first of these has several references from other pages; it might be the one to look at first, but it's really not a big deal in any case.

6.1.6. Initial Architecture Plans

LDAP offers the necessary management for identity resources, including regular POSIX account information for using l4tm from the shell prompt and storage of SSH public keys for non-password based authentication.

6.1.7. Future Architecture Plans

As the authentication service provides some fairly fundamental connection setup information for services within the machine, we'll want to implement some redundancy.

Accounts within the node environments themselves will be moved out of this authentication service and into a per-tenant authentication domain.

6.2. Authorization Service

The authorization service constructs a permissions "token" from authentication information, allowing this token to be passed around the system independently of the underlying authentication information and used to perform access control in services running within the machine environment.

This is especially useful in a RESTful environment, where the overhead of performing constant authentication and permissions queries back to the authentication service would be expensive.

6.2.1. Requirements

Use the Authentication Service to validate agent identities within the system. This means hooking up the system to LDAP.

Construct secure access control tokens which can be validated without needing to constantly communicate with the authentication system.

6.2.2. Initial Architectural Plans

Keystone looks like it offers what we need here, although it's quite heavy weight. It already has notional support for LDAP, although I haven't managed to make this work yet.

6.3. Service Connection Authentication

All connections within the machine management environment will use TLS, both to secure the information against disclosure and tampering, as well as to allow bi-directional authentication of the agents.

This means we need mechanisms for generating, storing and transmitting TLS keys among the participants.

6.3.1. Requirements

Generate or discover TLS keys for services and agents within the system.

Provide public and private keys to these services and agents.

6.3.2. Initial Architectural Plans

Key generation for services running on the management server will be generated as a part of service installation and configuration. We will need to have a list of where the public keys are installed to make them available across the whole system.

The public keys will be gathered together and incorporated into the

6.4. Login

While account data for users of the machine can be centralized in an LDAP database, we need some mechanism for offering suitable user accounts on individual nodes.

6.4.1. Requirements

- Provide global user identity
- Store shared user information(?)
- Offer home directories for each user

6.4.2. Initial Architecture Plans

Use the central LDAP service to hold user account information for nodes within the machine. This can be an opt-in process; nodes wanting central account management can use LDAP while others could perform local account management on their own.

Manifests for a node operating system will include the necessary pam configuration bits to use the central LDAP server for username and passwords.

Each node will mount the /home partition from the ToRMS using NFS to provide shared stable storage for small amounts of user information, such as shell customization and application binaries.

Each node will also mount the /srv paritition from the ToRMS, allowing groups to collaborate across the machine. Groups of users may have a directory in the /srv partition for their shared use.

For now, account creation will be managed by the LDAP administrator.

6.4.3. Future Architecture Plans

Tenants will need to be able to create and manage accounts within their own domains, so some kind of per-tenant LDAP database will eventually be needed.

6.5. Identity Services

With the global shared memory accessible by all nodes within a single load/store domain, control over access to that resource must be similarly global in scope. Individual nodes within the machine must not intrinsically have access to any of the central store, rather they must authenticate themselves within a global context and gain authorization to the store in a similar fashion.

An “identity” within the machine is secured with a public/private key pair. Authentication is performed by demonstrating control over a private key identified as being owned by that identity.

As with many mechanisms within the machine, the root of identity lies in the top-of-rack management server, at least in the current architecture. In the current design, this identity service is concerned strictly with OS access to FAM resources. Control over FAM encryption keys and access to services not related to FAM access are out of scope for the current effort.

6.5.1. Identities

To talk about identity and authentication within the machine context, we first need to define the set of agents needing identity.

1. Identity Agent. The root of identity management needs its own identity to authenticate communications with it.

2. Other global services on the ToRMS. These may simply share in a single ToRMS identity, but we may also decide to offer separate identity for each.

3. SoC hardware. This validates the connection between an executing OS instance and FAM access

4. MP identity. TSL public/private key using RSA

5. Psyloc identity

6. Tenants. These identify external principals permitted to access the machine and configure OS instances on nodes. Within the Librarian, each Shelf is owned by a Tenant.

Within the node hardware, we must ensure that the firewall controller access configuration is aligned with the operating system access to FAM regions. The Librarian must know that the firewall configuration commands are processed by the firewall agent which is resident on the same node as the OS instance making Shelf access requests. This is done by having communications with the Library File System and the Firewall Controller authenticated with the Node hardware key.

Within the ToRMS, a mapping between Node and Tenant is managed by the OS provisioning service. The Librarian uses this mapping to authorize access to Shelves.

6.5.2. Initial Implementation

Identity for the ToRMS services will live in files on that server, with each service directly accessing the secret key file necessary to authenticate themselves to other agents within the system.

The node identities will live in a file which is provisioned at OS installation time.

The authentication service will hold the public keys for all of the identities in the system. See Section 6.1, “Authentication Service” for a discussion on that service.

The public key for the authentication service itself must be provisioned during the OS installation and referenced from the ldap.conf file used by Node OS services.

6.5.3. Final implementation

Node identities will migrate to a secure location within the node, either with the TPM or using write-once fuses.

We should investigate using the Atalla box to help securely store identity information.

6.5.4. APIs

Storing public keys in LDAP will require a new LDAP schema.

We will probably want to provide convenience APIs on top of the standard LDAP libraries to make getting TLS keys for each service easier.

6.6. Assembly Agent

The Assembly Agent (AA) orchestrates and automates certain functions of the machine and is the primary route for managing changes to system configuration. The AA is a general-purpose management platform, capable of managing a variety of different systems. Each managed system is expressed using a different

Declarative Model (see below), perhaps conforming to a different Schema. For example, one schema may describe a system capable of deploying and managing Containers, while another schema may describe a large, complex, physical infrastructure. Its application is currently limited to managing the physical infrastructure of The Machine and its functions are also currently limited to:

- Fabric and memory power up
- Memory and fabric power down
- Assignment of tenant identity to SoCs
- Assignment of OS manifests to SoCs
- SoC power management
- System state repository

The Assembly Agent exposes a single REST API which enables interrogation or modification of the system state. All operations can be performed directly via that API or indirectly via the Management Dashboard.

Developers may modify SoC-OS assignments, applications may retrieve information on their host SoCs, and other management software can use it to coordinate global functions. For example, as part of its access control process, the Librarian can interrogate the Assembly Agent to determine the tenant identity associated with a SoC that has requested access to a Book.

The current state of the system is derived from the updates provided by the Monitoring Service which in turn collects data regarding:

- All hardware functions (excluding the SoCs) via Management Firmware (running on the Node MPs and Switch MPs)
- SoC state via its running Operating System
- Operating System state from the OS itself

Actions impacting hardware function interact directly with the REST APIs provided by the MPs. Updates to SoC-OS assignments are passed to the OS Manifesting Service via its REST API.

The Assembly Agent sits on both the Management and System networks.

6.6.1. Declarative Management

The Assembly Agent adopts a style of operation known as Declarative Management, where the desired configuration of the system under management is expressed using a Declarative Model known as the Desired State. The Desired State encodes the desired structure and properties of the system under management. The role

of the AA is to reconfigure the managed system to achieve and maintain the Desired State, by performing a sequence of actions (the Plan) on the system under management. The AA continually monitors the Current State of the managed system, and compares it with the Desired State. The appropriate Plan, consisting of a sequence of actions, to achieve the Desired State from the Current State is automatically computed by the AA via an automated planning process. If during execution of the plan some actions fail or if, subsequent to reaching the Desired State, the system deviates from this state (e.g. due to failures), the AA will automatically re-plan to create a new sequence of actions to reach or maintain the Desired State, respectively. Likewise, if the Desired State changes, the AA will compute a new plan to reach the new Desired State.

By contrast, an Imperative Management needs to be explicitly given the sequence of actions to be performed on the managed system to achieve the required configuration. If failures occur, the management system cannot easily automatically adapt. Instead, it needs to be explicitly given a new sequence of actions to perform.

6.6.2. Dependencies

- Authentication Service
- Authorization Service
- Monitoring Service
- OS Manifesting Service
- Management Firmware

6.6.3. Initial Architecture Plans

The Assembly Agent is centralised and runs on the top-of-rack management server.

6.6.4. Future Architecture Plans

The Assembly Agent is intended to be a distributed capability, ultimately with agents residing on each MP. The next intermediate step to this goal is for this distributed architecture to still be hosted centrally on the management server to prove its validity before dealing with the issues of running in the resource-constrained MP environment.

6.6.5. API

The AA runs within an HTTP server that accepts HTTP connections to receive requests and perform operations via a REST interface. This outward interface has the following qualities:

- HTTP/S 1.1 (TLS)
- Representation State Transfer (REST)
- JavaScript Object Notation (JSON) is used to exchange models and state
- UTF-8 encoding

Based on the above qualities, every PUT and PATCH request should have the following HTTP header, which describes the format of the data contained in the body of the request:

- Content-Type: application/json; charset=utf-8

The interface supports four HTTP methods: GET, PUT, PATCH, and DELETE. This document describes the external API of the AA, using these methods. Unless otherwise stated, the URLs supported by the AA will work with or without a trailing slash (“/”); for example, both the URI “/desired” and “/desired/”are equivalent.

Although the methods described in this document show the possibility of the HTTP response code “401 Unauthorized”, the AA does not yet support authorization; the intent is to add support for authorization in a future release.

Note that this document only describes the REST methods used to manipulate and retrieve the various Declarative Models used by the AA. It does not describe the schema and details of specific models for specific systems managed by the AA. A separate document will describe the specific model and schema for The Machine.

There are three models used by the AA to manage a system – the Desired State, the Expanded Desired State, and the Current State. All three are defined using the JSON representation of the SFP modeling language, described in the Assembly Agent Model Language Documentation [1], and can be retrieved from the AA by a GET request. However, only the Desired State can be directly set or modified via the REST API, via PUT, PATCH, and DELETE methods. When retrieving or manipulating the JSON models that represent the various states, most of the APIs allow a specific location within the JSON model to be referenced via a path parameter, specified using a string containing a JSON pointer.

6.6.6. Desired State

The Desired State model describes the goal state of the managed system. It is defined using the JSON representation of the SFP modeling language, described in footnotref:[aa-model]. The Desired State model, also known as the Unexpanded Desired State (UDS), goes through a compilation process to create the Expanded Desired State (EDS). During compilation, the AA will, for example, process any

[1] “Assembly Agent Model Language Documentation”.

include statements in the Desired State to include files containing Schema referenced in the Desired State, and process references between attributes. It is the resulting EDS that the AA actually uses as the goal when comparing with the Current State in order to create an appropriate plan to achieve that goal. See the section called “SET Desired State” for an example Desired State that includes a Schema file, while the section called “GET Expanded Desired State” shows the resulting EDS after compilation.

SET Desired State

Description

Set the Desired State of the AA at the specified path.

Base Endpoint

“/desired” or “/desired/{path}”

HTTP Method

PUT

Input

path: (Optional) Path of JSON node to add or replace in existing Desired State. If path is not specified, the path is the root location in the JSON representation of the existing Desired State in the AA.

Body

The body of the PUT request contains the UDS model. The body is of the form:

```
{
        "value": {
                ... Desired State model
        }
}
```

An example body containing a Desired State model is shown below:

```
{
        "value": {
                "include": [
                        "/rest/simplenodetest/simple_node_schema.json"
                ],
                "main": {
                        "node1": {
                                "fwVersion": "v2",
                                "installedFwVersion": "v2",
                                "power": "$(power.on)",
                                "type": "Node"
```

```
                        }
                }
        }
}
```

Response

200 OK, *201 Created*

If the request is successful, or only contains warnings, the response is a CompilationResult, containing information about the outcome of the compilation process. The format of CompilationResult is:

```
{
        "type":"One of OK or WARNING",
        "message":"Human-readable compilation message"
}
```

For example,

```
{
        "type":"OK",
        "message":"OK"
}
```

The response code will be “200 OK” if JSON ***replace*** semantics are followed, and “201 Created” if JSON Patch ***add*** semantics are followed.

Errors

400 Bad Request: HTTP Response if a compilation error occurred, or the body contained invalid JSON.

401 Unauthorized: HTTP Response if caller is not authorised to perform the operation.

404 Not Found: HTTP Response if the specified path does not exist.

If an error occurs, the desired State will not be modified as a result of the operation. More details about the error may be returned in the body of the response in an ErrorMessage. The format of ErrorMessage is:

```
{
        "status":"HTTP Response Code",
        "message":"Human-readable error message"
}
```

For example,

```
{
        "status":"404",
        "message":"Path does not exist"
}
```

Example

```
curl -XPUT -d @aa_desired.json -H "Content-Type: application/json" http://aa.hpe.com/dma/
desired
```

Detailed Description

The PUT method adds or replaces the Desired State at the specified path with the specified new value. There are three possibilities for the node referenced at the specified path, each with a correspondence to JSON Patch semantics (see RFC 6902 [2] and PATCH Desired State operation below).

Firstly, if a node already exists at the specified path, it is simply replaced with the new value, using replace JSON Patch semantics. For example, if the existing Desired State was the JSON document `{"main":{"list":["An entry"]}}`, and a PUT operation performed to “/desired/main/list“, containing the body `{"value":{"foo":{"bar":"A value"}}}`, then the array at “list” would be replaced with a JSON container node, resulting in the Desired State `{"main":{"list":{"foo":{"bar":"A value"}}}}`.

Secondly, if a node at the specified path does not exist, and the immediate parent node of the specified path is a JSON container node, then a new node is created in the parent JSON container with the specified value, using add JSON Patch semantics. For example if a node at "/a" does not yet exist and the method is called at the path "/a" with a container node value such as `{"b":42}`, then, because the parent of “/a” (i.e. “/”) exists and is a container, a container node called "a" will be created at "/" that contains `{"b":42}`; if the desired state was originally the empty JSON container, {}, then the resulting document would be `{"a":{"b":42}}`.

Thirdly, if the last element of the path is "-", the path is assumed to refer to the last element of an array, and the parent element in the path must to refer to an array. In this case, the set operation is equivalent to an add operation in JSON Patch, where the specified new value is added to the end of the array. For example, if the existing Desired State was the JSON document `{"main":{"list":["An entry"]}}`, and a PUT operation performed to “/desired/main/list/-“, containing the body `{"value":"another"}`, the resulting Desired State would be `{"main":{"list":["An entry","another"]}}`.

If none of the above cases for the referenced path are true, an HTTP 404 Not Found response is returned. For example, an operation to PUT a value to “/a/b/c” when only “/a” is an existing container, but “/a/b” does not exist or is not a container, would result in HTTP 404.

The new value contained in the body can either be a JSON value node (such as an Integer or String), or a container node. If the new value is a value node, and

[2] “IETF RFC 6902 - JavaScript Object Notation (JSON) Patch”

the old value at the referenced path is a container node, then the effect of the replacement with the new value will be to delete the sub-tree at the specified path and replace it with the Value Node.

PATCH Desired State

Description

Modify the Desired State of the AA using the list of operations specified in a JSON Patch document contained in the body of the request. The patch operation can perform multiple modifications to the desired state, using the set of patch methods as defined in RFC 6902 [2].

Base Endpoint

"/desired"

HTTP Method

PATCH

Body

The body of the PATCH request contains the a JSON document conforming to the JSON Patch structure, described in RFC 6902 [2], that defines a list of add, remove, and delete operations to be applied to the Desired State. For example,

```
[
        {"op":"add","path":"/main/name","value":"Foo"}
]
```

Response

200 OK

If the request is successful, or only contains warnings, the response is a CompilationResult, containing information about the outcome of the compilation process. See the section called "SET Desired State" for a description of CompilationResult.

Errors

400 Bad Request: HTTP Response if a compilation error occurred, or a badly formed patch JSON document was supplied.

401 Unauthorized: HTTP Response if caller is not authorised to perform the operation.

If an error occurs, the desired State will not be modified as a result of the operation. More details about the error may be returned in the body of the response in an *ErrorMessage*. See the section called "SET Desired State" for a description of *ErrorMessage*.

Example

```
curl -XPATCH -d @aa_patch_desired.json -H "Content-Type: application/json" http://
aa.hpe.com/dma/desired
```

Detailed Description

The PATCH method applies all of the operations defined in the JSON document supplied in the body to the existing Desired State. For example, if the existing Desired State was the JSON document `{"main":{"list":["An entry"]}}`, and a PATCH operation performed that contained the body shown above, then a new value at "/main/name" would be added to the JSON container node at "/main", resulting in the Desired State `{"main":{"list":["An entry"],"name":"Foo"}}`.

DELETE Desired State

Description

Delete the node in the Desired State of the AA at the specified path.

Base Endpoint

"/desired" , "/desired/{path}"

HTTP Method

DELETE

Input

path: (Optional) Path of JSON node to delete. If path is not specified, the path is the root location in the JSON representation of the existing Desired State in the AA, and the effect is to clear the Desired State.

Response

200 OK

If the request is successful, or only contains warnings, the response is a CompilationResult, containing information about the outcome of the compilation process after the deletion. See the section called "SET Desired State" for a description of CompilationResult.

Errors

400 Bad Request: HTTP Response if a compilation error occurred.

401 Unauthorized: HTTP Response if caller is not authorised to perform the operation.

404 Not Found: HTTP Response if the specified path does not exist.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called “SET Desired State” for a description of *ErrorMessage*.

Example

```
curl -XDELETE http://aa.hpe.com/dma/desired/name
```

Detailed Description

The DELETE method applies all of the operations defined in the JSON document supplied in the body to the existing Desired State. For example, if the existing Desired State was the JSON document `{"main":{"list":["An entry"],"name":"Foo"}}`, and a DELETE operation performed on “/main/name” then the referenced JSON node would be removed from the JSON Container at “/main”, resulting in the Desired State `{"main":{"list":["An entry"]}}`.

GET Desired State

Description

Get the Desired State of the AA at the specified path.

Base Endpoint

“/desired”, “/desired/{path}”

HTTP Method

GET

Input

path: (Optional) Path of JSON node to retrieve. If path is not specified, the path is the root location in the JSON representation of the existing Desired State in the AA.

Response

200 OK: If the request is successful, the Desired State model at the specified path is returned in the body of the response. The response body is of the form:

```
{
        "value": {
                ... Returned Desired State model
        }
}
```

The “value” key is used to hold the returned JSON value from the location in the state referenced by path. The value could be any valid JSON node, such as a primitive type (such as Integer or String), an array, or a container. Note that

"value" is not used when forming a path to reference a node within the JSON representation of the state.

Errors

401 Unauthorized: HTTP Response if caller is not authorised to perform the operation.

404 Not Found: HTTP Response if the specified path does not exist.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called "SET Desired State" State for a description of *ErrorMessage*.

Example

```
curl -XGET http://aa.hpe.com/dma/desired
```

Detailed Description

Assuming the Desired State set in the section called "SET Desired State", a GET on "/desired/" would return the complete Desired State:

```
{
        "value": {
                "include": [
                        "/rest/simplenodetest/simple_node_schema.json"
                ],
                "main": {
                        "node1": {
                                "fwVersion": "v2",
                                "installedFwVersion": "v2",
                                "power": "$(power.on)",
                                "type": "Node"
                        }
                }
        }
}
```

While a GET on "/desired/main/node1/type" would return the value of an attribute:

```
{
        "value":"Node"
}
```

A GET on "/desired/include" would return the complete array:

```
{
        "value": [
                "/rest/simplenodetest/simple_node_schema.json"
        ]
}
```

While a GET on "/desired/include/0" would return first element of the array:

```
{
        "value": "/rest/simplenodetest/simple_node_schema.json"
```

```
}
```

6.6.7. Expanded Desired State

The Expanded Desired State (EDS) model describes the goal state of the managed system. It is created within the AA by compiling the Desired State set by the user, and is described using the JSON representation of the SFP modeling language, detailed in the model language [1]. It is the EDS that the AA uses as the goal when comparing with the Current State in order to create an appropriate plan to achieve that goal.

GET Expanded Desired State

Description

Get the Expanded Desired State (EDS) of the AA at the specified path.

Base Endpoint

“/eds”, “/eds/{path}”

HTTP Method

GET

Input

path: (Optional) Path of JSON node to retrieve. If path is not specified, the path is the root location in the JSON representation of the Expanded Desired State in the AA.

Query Parameters

The following optional query parameters can be specified.

Filter parameters

The following Boolean query parameters allow various forms of filtering on the information returned for the EDS. If not specified, the values default to false, so no filtering will be applied.

`filteralways`

If set to true, filter "always" elements from the returned EDS.

`filtertypes`

If set to true, filter "types" elements from the returned EDS.

`filteractions`

If set to true, filter "actions" elements from the returned EDS.

`filtervariableattribs`

If set to true, filter "variable" attribute elements from the returned EDS.

`filterconstantattribs`

If set to true, filter constant attribute elements from the returned EDS.

Response

200 OK: If the request is successful, the Expanded Desired State model at the specified path is returned in the body of the response. The response body is of the form:

```
{
  "value": {
    ... Returned Expanded Desired State model
  }
}
```

For example, a GET on “/eds/” might return the Expanded Desired State (compiled from the Desired State):

```
{
  "value": {
    "types": {
      "power": {
        "type": "enum",
        "symbols": [
          "off",
          "on"
        ]
      },
      "Node": {
        "type": "schema",
        "extends": [
          "schema"
        ]
      }
    },
    "node1": {
      "type": "Node",
      "stop": {
        "type": "action",
        "local_condition": [
          "=",
          "node1.observed_power",
          "on"
        ],
        "effect": {
          "node1.power": "off",
          "node1.observed_power": "off"
        }
      },
      "start": {
        "type": "action",
        "parameters": {
          "v": "string"
        },
```

```
      "local_condition": [
        "and",
        [
          "=",
          "node1.observed_installedFwVersion",
          "$(v)"
        ],
        [
          "=",
          "node1.observed_power",
          "off"
        ]
      ],
      "effect": {
        "node1.power": "on",
        "node1.observed_power": "on",
        "node1.observed_fwVersion": "$(v)",
        "node1.fwVersion": "$(v)"
      }
    },
    "power": {
      "variable": true,
      "value": "on",
      "enum": "power"
    },
    "observed_power": {
      "variable": true,
      "value": "on",
      "enum": "power"
    },
    "observed_installedFwVersion": {
      "variable": true,
      "value": "v2",
      "type": "string"
    },
    "observed_fwVersion": {
      "variable": true,
      "value": "v2",
      "type": "string"
    },
    "installedFwVersion": {
      "variable": true,
      "value": "v2",
      "type": "string"
    },
    "installFw": {
      "type": "action",
      "parameters": {
        "v": "string"
      },
      "effect": {
        "node1.observed_installedFwVersion": "$\n\t\t\t\t\t\t(v)",
        "node1.installedFwVersion": "$(v)"
      }
    },
    "fwVersion": {
      "variable": true,
      "value": "v2",
      "type": "string"
    }
  }
```

```
  }
}
```

While a GET on “/eds/node1/fwVersion” would return:

```
{
  "value": {
    "variable": true,
    "value": "v2",
    "type": "string"
  }
}
```

Errors

401 Unauthorized: HTTP Response if caller is not authorised to perform the operation.

404 Not Found: HTTP Response if the specified path does not exist.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called “SET Desired State”> for a description of *ErrorMessage*.

Example

```
curl -XGET http://aa.hpe.com/dma/eds
```

To return only the attributes within the EDS:

```
curl -XGET http://aa.hpe.com/dma/eds?filteralways=true&
filtertypes=true&filteractions=true
```

6.6.8. Current State

The Current State describes the current state of the managed system. It is maintained within the AA by monitoring the actual state of the managed system. Like all AA models, it is described in JSON using the SFP modeling language, detailed in [1]. In response to changes in either desired state or current state, the AA will automatically create plans to perform an appropriate sequence of actions to change the Current State of the managed system to be equivalent to the Expanded Desired State, in order to maintain the Desired State. Equivalence between Expanded Desired State and Current State is defined by comparing the values of nodes in the EDS that represent variables (have the key "variable" set to the value true), with the corresponding values in the Current State. Any schema, types, or actions defined in the EDS are ignored in the comparison. In the EDS example above, the variables “node1/fwVersion”, “node1/installedFwVersion”, “node1/observed_installedFwVersion”, “node1/installedFwVersion”, “node1/observed_installedFwVersion”, “node1/power”, and “node1/observed_power” will be compared with the corresponding elements in the Current State.

GET Current State

Description

Get the Current State of the AA at the specified path.

Base Endpoint

“/current”, “/current/{path}”

HTTP Method

GET

Input

path: (Optional) Path of JSON node to retrieve. If path is not specified, the path is the root location in the JSON representation of the Current State in the AA.

Response

200 OK

If the request is successful, the Current State at the specified path is returned in the body of the response. The response body is of the form:

```
{
  "value": {
    ... Returned Current State
  }
}
```

For example, a GET on “/current/” might return the Current State:

```
{
  "value": {
    "node1": {
      "type": "Node",
      "power": "on",
      "observed_power": "on",
      "observed_installedFwVersion": "v2",
      "observed_fwVersion": "v2",
      "installedFwVersion": "v2",
      "fwVersion": "v2"
    }
  }
}
```

While a GET on “/current/node1/power” would return: { "value": "on" }

Errors

401 Unauthorized: HTTP Response if callers is not authorised to perform the operation.

404 Not Found: HTTP Response if the specified path does not exist.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called “SET Desired State” for a description of *ErrorMessage*.

Example

```
curl -XGET http://aa.hpe.com/dma/current
```

6.6.9. Plans

In response to changes in either Desired State or Current State, the AA automatically creates and executes plans to bring the state of the managed system to the goal state. These plans can be retrieved from the AA, in order to see how the AA achieved the goal state. There are two sets of plans, Active Plans and Archived Plans, each containing a list of plans. Active Plans are plans that are either waiting to be scheduled, or are currently being executed. Archived Plans are plans that have completed, because they executed successfully, failed, or were canceled.

When a change occurs to the Desired State, the AA may need to cancel existing Active Plans if the change conflicts with the effects of those plans. For example, if a change occurs to a Desired State variable and that variable will be affected by an Active Plan, then the existing plan will be canceled and a new plan generated. However, if there is no overlap between the effects of Active Plans and the effect of any new plan required to achieve changes in the Desired State, then the existing Active Plans can be allowed to run to completion.

Note

The detailed representation of Plans is not currently described in this document, because the format is subject to change.

GET Active Plans

Description

Get the list of Active Plan IDs in the AA.

Base Endpoint

“/plans/active”

HTTP Method

GET

Response

200 OK: If the request is successful, the list of IDs of Active Plans is returned in the body of the response. The response body is of the form:

```
["planid1","planid2", …]
```

For example, a GET on “/plans/active” might return:

```
["99d6ac67-49ec-440b-a9df-38a43b8527f9"]
```

Errors

401 Unauthorized: HTTP Response if callers is not authorised to perform the operation.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called “SET Desired State” for a description of *ErrorMessage*.

Example

```
curl -XGET http://aa.hpe.com/dma/plans/active
```

GET Active Plan

Description

Get a specific Active Plan in the AA, identified by its ID.

Base Endpoint

“/plans/active/{id}”

HTTP Method

GET

Input

id: ID of the plan.

Response

200 OK: If the request is successful, the details of the Active Plan are returned in the body of the response. For example, a GET on “/plans/active/99d6ac67-49ec-440b-a9df38a43b8527f9” might return:

```
{
  "state": "COMPLETE",
  "replanNeeded": false,
  "planId": "99d6ac67-49ec-440b-a9df-38a43b8527f9",
  "plan": {
    "version": 1,
    "type": "parallel",
    "initial_actions": [
      0
    ],
    "active_variables": {
      "node1.power": 1,
      "node1.observed_power": 1,
```

```
    "node1.observed_installedFwVersion": 1,
    "node1.observed_fwVersion": 1,
    "node1.fwVersion": 1
  },
  "actions": [
    {
      "variables": [
        "node1.observed_installedFwVersion",
        "node1.observed_fwVersion",
        "node1.power",
        "node1.observed_power",
        "node1.fwVersion"
      ],
      "timeout": 120,
      "remote_verified": null,
      "remote_condition": null,
      "post_execution": null,
      "parameters": {
        "v": {
          "value": "v2",
          "type": "string"
        }
      },
      "object": {
        "type": "Node",
        "path": "node1"
      },
      "method": "start",
      "local_verified": null,
      "local_condition": {
        "node1.observed_power": {
          "value": "off",
          "type": "enum power"
        },
        "node1.observed_installedFwVersion": {
          "value": "v2",
          "type": "string"
        }
      },
      "id": 0,
      "effect": {
        "node1.power": {
          "value": "on",
          "type": "enum power"
        },
        "node1.observed_power": {
          "value": "on",
          "type": "enum power"
        },
        "node1.observed_fwVersion": {
          "value": "v2",
          "type": "string"
        },
        "node1.fwVersion": {
          "value": "v2",
          "type": "string"
        }
      },
      "before": [],
      "after": []
    }
```

```
      ]
    },
    "cancellationRequested": false,
    "actionExecutions": {
      "a0": {
        "state": "SUCCESS",
        "id": 0
      }
    }
  }
```

Errors

401 Unauthorized: HTTP Response if callers is not authorised to perform the operation.

404 Not Found: HTTP Response if the specified plan ID does not exist in the set of Active Plans.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called “SET Desired State” for a description of *ErrorMessage*.

Example

```
curl -XGET http://aa.hpe.com/dma/ plans/active/99d6ac67-49ec-440ba9df-38a43b8527f9
```

GET Archived Plans

Description

Get the list of Archived Plan IDs in the AA.

Base Endpoint

“/plans/archived”

HTTP Method

GET

Response

200 OK: If the request is successful, the list of IDs of Archived Plans is returned in the body of the response. The response body is of the form:

```
["planid1","planid2", …]
```

For example, a GET on “/plans/archived” might return:

```
["99d6ac67-49ec-440b-a9df-38a43b8527f9"]
```

Errors

401 Unauthorized: HTTP Response if callers is not authorised to perform the operation.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called “SET Desired State” for a description of ErrorMessage.

Example

```
curl -XGET http://aa.hpe.com/dma/plans/archived
```

GET Archived Plan

Description

Get a specific Archived Plan in the AA, identified by its ID.

Base Endpoint

“/plans/archived/{id}”

HTTP Method

GET

Input

id: ID of the plan.

Response

200 OK: If the request is successful, the details of the Archived Plan is returned in the body of the response. For example, a GET on “/plans/archived/99d6ac67-49ec-440b-a9df38a43b8527f9” might return a plan of a similar form to that shown in the section called “GET Active Plan”.

Errors

401 Unauthorized: HTTP Response if callers is not authorised to perform the operation.

404 Not Found: HTTP Response if the specified plan ID does not exist in the set of Archived Plans.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called “SET Desired State” for a description of *ErrorMessage*.

Example

```
curl -XGET http://aa.hpe.com/dma/ plans/archived/99d6ac67-49ec-440ba9df-38a43b8527f9
```

6.6.10. Status

The Assembly Agent maintains a status of the state of the system. Currently this status is very simple, detailing the software build of the running server. Over time this status will become more sophisticated, returning a richer set of information about the internal state of the AA.

GET Status

Description

Get the current Status of the AA.

Base Endpoint

"/status"

HTTP Method

GET

Response

200 OK: If the request is successful, the current Status of the AA is returned in the body of the response. The Status is currently of the form:

```
{
  "version": "release version of the build",
  "name": "name of server"
}
```

For example, a GET on "/status" might return the status:

```
{
  "version": "0.1-SNAPSHOT (11f31bce)",
  "name": "dma-server"
}
```

Errors

401 Unauthorized: HTTP Response if caller is not authorised to perform the operation.

If an error occurs, more details about the error may be returned in the body of the response in an ErrorMessage. See the section called "SET Desired State" for a description of *ErrorMessage*.

Example

```
curl -XGET http://aa.hpe.com/dma/status
```

6.6.11. API Summary

Table 6.1. API Summary

	Subgroup	Description	URI	HTTP
STATE	*Desired State*	Add or replace Desired State at specified path, using JSON model supplied in body	`/desired, /desired/{path}`	PUT
		Modify Desired State using specified JSON Patch operations supplied in body	`/desired`	PATCH
		Delete Desired State at specified path	`/desired, /desired/{path}`	DELETE
		Get Desired State at specified path	`/desired, /desired/{path}`	GET
	Expanded Desired State	Get Expanded Desired State at specified path	`/eds, /eds/{path}`	GET
	Current State	Get Current State at specified path	`/current, /current/{path}`	GET
PLANS	*Active Plans*	Get list of active plan IDs	`/plans/active`	GET
		Get a specific active plan	`/plans/active/{planId}`	GET
	Archived Plans	Get list of archived plan IDs	`/plans/archived`	GET
		Get a specific archived plan	`/plans/archived/{planId}`	GET
STAT	*Assembly Agent Status*	Get AA Status	`/status`	GET

	Subgroup	Description	URI	HTTP
US				

6.7. Monitoring Service

The Monitoring Service is the primary mechanism for capturing, recording and propagating data between Management Firmware, Management Software, Operating System instances and applications. Data can take three forms: logs, events and metrics. Log data represents the messages generated by the source, have little in the way of standard structure across all senders and are generally intended for human consumption. Event data is structured for easy parsing by software and is used to indicate important changes in the state of the source. Metric data is similar to event data in that is highly structured but it is always numerical in nature and usually generated at a much higher rate than log or event data.

The Monitoring Service presents a REST API that can be used to record data and respond to queries from any authorised entity. Log and event data are stored in a SQL database while metric data is stored in a separate database optimised for time series storage and retrieval.

Dedicated Agents (based on Monasca agents) are used to support data capture from software components that were not built to integrate with the Management Service. These Agents poll their assigned source to determine changes in state and then forward that information (structured as log, event or metric data) to the Monitoring Service REST API. Those components that have been designed to work with the Monitoring Service may simply, and more efficiently, push updates directly to the Monitoring Service without the need for dedicated Agents.

The current hardware state is captured from Management Firmware via the MPs' REST APIs with the exception of SoC state which is not visible to the Node MP (other than its power status). All MPs use a single Agent that will run on the management server. Each Operating System instance will run an Agent locally that will capture information (via StatsD and logstash) on both its own function and the SoC.

Management Software and applications may also push data to the Monitoring Service for later analysis but there is no requirement for them to do so.

The Monitoring Service sits on both the Management and System networks.

6.7.1. Dependencies

- Authentication Service
- Authorisation Service

- Complete data collection requires:
- Management Firmware Agents (Node MPs, Switch MPs)
- Management Software Agents
- Operating System Agents

6.7.2. Initial Architecture Plans

The dedicated Agents for each MP will be retired when Management Firmware has been modified to push their updates to the Monitoring Service directly.

The number of Agents used for MP collection may be scaled up depending on load.

6.7.3. Future Architecture Plans

Should performance of the API prove inadequate for queries returning large amounts of information, especially metrics, then an alternative will be provided for use in those situations.

The addition of a publish/subscribe interface will provide a more efficient way of updating subscribers, e.g. the Management Dashboard.

When persistent memory is available the Monitoring Service will be retired and replaced with a more suitable solution.

6.7.4. API

The single REST API presented by the Monitoring Service is the only mechanism for publishing and querying data.

6.7.5. Metrics

GET Metrics

Description

Gets a set of measurement metrics according to the specified parameters.

Base Endpoint

“/rest/metrics”

HTTP Method

GET

Input

startTime (Required)

The start time for the query as number of milliseconds since 1970-01-01 00:00:00 UTC.

metric (Required)

Metric name of the metric data to be retrieved.

endTime (Optional)

The end time for the query as the number of milliseconds since 1970-01-01 00:00:00 UTC.

filter (Optional)

A general filter that narrows the list of resources returned. Can be more than one (i.e filter=x=1&filter=y=2).

coordinate (Optional)

A specific filter that narrows the list of resources returned based on a section of the resource coordinate. (i.e for coordinate machine_rev/1/datacenter/1 the filter is coordinate=machine_rev=1&coordinate=datacenter=1)

Response

200 OK

Metric List. The array of filtered metrics from startTime to endTime.

Examples

Get all measurements from a metric:

Request

```
https://{hostname}/rest/metrics?startTime=1445279466&metric=mp.net.avg.RxBandwidth
```

Response


```
{
  "starttime": 1445279466,
  "endtime": 1445279902,
  "metrics": {
    "name": "mp.net.avg.RxBandwidth",
    "columns": [
```

```
          "timestamp",
          "hostname",
          "value"
        ]
      },
      "values": [
        [
          "2015-10-30T19:17:09Z",
          "myhost",
          33554432
        ],
        [
          "2015-10-30T19:18:09Z",
          "myhost",
          345333444
        ]
      ]
    }
```

Get all metric data from a time range:

Request

```
https://{hostname}/rest/metrics? startTime=1445279466&metric=mp.net.avg.RxBandwidth&
endTime=1445279902&filter=hostname=myhost
```

ADD Metrics

Description

Adds a set of metrics based upon the attributes specified. All attributes without default values must be specified in the POST body.

Base Endpoint

“/rest/metrics”

HTTP Method

POST

Body

The body of the PUT request contains the MetricData to add. The body is of the form:

```
[
  {
    "timestamp": timestamp,
    "dimensions": {
      "hostname": hostname
    },
    "name": metric_name,
    "value": metric_value
  }
```

```
]
```

Response

201 Created

Success on creating the metrics.

Examples

Add a set of metrics with default values: `https://{localhost}/rest/metrics`

RequestBody

```
[
  {
    "timestamp": 1443575142000,
    "dimensions": {
      "hostname": "myhost"
    },
    "name": "load.avg_1_min",
    "value": 0
  },
  {
    "timestamp": 1443575142000,
    "dimensions": {
      "hostname": "myhost"
    },
    "name": "load.avg_15_min",
    "value": 0.05
  }
]
```

6.7.6. Logs

This API allows you to get the logs from a specified source, and create new logs from sources.

LIST logs

Description

Retrieves a list of logs from specified source and by a time range.

Base Endpoint

“/rest/logs”

HTTP Method

GET

Input

startTime (Required)

The start time is of type timestamp (epoch time).

source (Required)

Source parameter that specifies the source name of the data (i.e. application). Can be more than one (i.e. source=mysource1,mysource2,mysource3).

endTime (Optional)

The end time for the query as the number of milliseconds since 1970-01-01 00:00:00 UTC.

filter (Optional)

A general filter that narrows the list of resources returned. Can be more than one (i.e filter=x=1&filter=y=2).

coordinate (Optional)

A specific filter that narrows the list of resources returned based on a section of the resource coordinate. (i.e for coordinate machine_rev/1/datacenter/1 the filter is coordinate=machine_rev=1&coordinate=datacenter=1)

Response

200 OK

If the request is successful, the response is a LogList containing a Collection representing the set of Logs from a specified source. The format of LogList is:

```
{
  "starttime": starttime,
  "endtime": endtime,
  "logMessages": [
    {
      "timestamp": timestamp
      "source": "source",
      "hostname": "hostname",
      "level": "log level",
      "message": "message"
    }
  ]
}
```

Examples

List logs starting at a specified time from apache:

Request

```
https://{hostname}/rest/logs?startTime=1445279466&source=apache
```

Response

```
{
  "startime": 1445279466,
  "endtime": 1445279902,
  "logMessages": [
    {
      "timestamp": 1445279466,
      "source": "apache",
      "message": "Any message here",
      "hostname": "myhost",
      "level": "INFO"
    },
    {
      "timestamp": 1445279469,
      "source": "apache",
      "message": "Any error message here",
      "hostname": "myhost",
      "level": "ERROR"
    }
  ]
}
```

Get all logs from a time range, specific host, and log level:

Request

```
https://{hostname}/rest/logs?startTime=1445279466&source=apache&
filter=hostname=myhost&filter=level=INFO&endTime=1445279902
```

Response

```
{
  "startime": 1445279466,
  "endtime": 1445279902,
  "logs": [
    {
      "timestamp": 1445279466,
      "source": "apache",
      "message": "Any message here",
      "hostname": "myhost",
      "level": "INFO"
    },
    {
      "timestamp": 1445279469,
      "source": "apache",
      "message": "Any error message here",
      "hostname": "myhost",
      "level": "ERROR"
    }
```

```
    ]
}
```

ADD log data

Description

Adds log data based upon the attributes specified. All attributes without default values must be specified in the POST body.

Base Endpoint

“/rest/logs”

HTTP Method

POST

Body

The body of the request contains the Log to add. The body is of the form:

```
{
        attribute: value,
        ...
}
```

Response

201 Created

Success on creating the log.

Examples

Add a log with default values:

Request

```
https://{localhost}/rest/logs
```

Body

{ "timestamp": 1445279466, "source": "apache", "message": "this is a log message", "hostname": "myhost", "level": "INFO", "coordinate": "datacenter/x/" }

6.7.7. Events

This API allows you to get the events from a specified source, and create new events from sources.

LIST events

Description

Retrieves a list of events from specified source and by a time range.

Base Endpoint

``/rest/events

HTTP Method

GET

Input

startTime (Required)

The start time is of type timestamp (epoch time).

source (Required)

Source parameter that specifies the source name of the event (i.e. application). Can be more than one (i.e. source=mysource1,mysource2,mysource3).

endTime (Optional)

The end time for the query as the number of milliseconds since 1970-01-01 00:00:00 UTC.

filter (Optional)

A general filter that narrows the list of resources returned. Can be more than one (i.e filter=x=1&filter=y=2).

coordinate (Optional)

A specific filter that narrows the list of resources returned based on a section of the resource coordinate. (i.e for coordinate machine_rev/1/datacenter/1 the filter is coordinate=machine_rev=1&coordinate=datacenter=1)

Response

200 OK

If the request is successful, the response is a Collection representing the set of events from a specified source. See Event object.

Examples

List events starting at a specified time from "source":

Request

```
https://{hostname}/rest/events?startTime=1445279466&source=source
```

Response

```
{
  "starTime": 1445279466,
  "endTime": 1445279902,
  "events": [
    {
      "timestamp": 1422568542600,
      "source": "source",
      "eventId": "7559",
      "eventName": "name",
      "eventType": "eventtype2",
      "coordinate": "/node/3004"
    },
    {
      "timestamp": 1422568542600,
      "source": "source",
      "eventId": "7560",
      "eventName": "name2",
      "eventType": "eventtype2",
      "coordinate": "/node/3005"
    }
  ]
}
```

Get all events from a time range, specific source, and event type:

Request

```
https://{hostname}/rest/events?startTime=1445279466&
filter=source=origin1&filter=eventType=StatusChange&endTime=1445279902
```

ADD event

Description

Adds an event based upon the attributes specified. All attributes without default values must be specified in the POST body.

Base Endpoint

``/rest/events

HTTP Method

POST

Body

The body of the request contains the Event to add. The body is of the form:

```
{
        attribute: value,
        ...
}
```

Response

201 Created

Success on creating the event.

Examples

Add an event with default values:

Request

```
https://{localhost}/rest/events
```

Body

{ "timestamp": 1422568542600, "source": "source", "eventId": "7560", "eventData": "On", "eventName": "name2", "eventType": "eventtype2", "coordinate": "/node/3005" }

6.7.8. Delete

This feature should only be available for the debugging and testing version of the monitoring service. Deletes all data for the specified resource.

DELETE resource

Description

Deletes all data of a specified resource.

Base Endpoint

“/rest/delete”

HTTP Method

DELETE

Input

type (Required)

The type(s) of resource(s) to delete.Options are log, event and metric. More than one option can be provided comma separated.

Response

200 OK

All content of the specified resource is deleted.

Examples

Delete log and event resources

Request

```
https://{localhost}/rest/delete?type=log,event
```

6.8. Management Dashboard

The Management Dashboard provides a graphical user interface that both presents a view on the current system state and exposes various management functions that may be invoked by the user at any time.

The projected system state is an aggregation of information obtained via the Assembly Agent, Monitoring Service and Librarian that includes:

- Hardware configuration, state and metrics with respect to fabric, memory and SoCs
- Memory management – Shelves, Books and their mapping to hardware components
- Operating System assignments – which OS manifest is in use by which SoC
- Tenant assignments – which SoC has been assigned to which Tenant

All users must login using a username and password before access is granted to the dashboard. In terms of access control, only two roles are currently supported: Tenant Administrator (full access) and Tenant Provider Administrator (everyone else – limited access).

The management functions that are exposed via the dashboard are serviced by the Assembly Agent include:

- SoC power management
- Operating System Manifest assignment to SoCs
- Tenant assignment to SoCs (Tenant Administrators only)
- SoC, memory and fabric power up and down (Tenant Administrators only)

Note that although the fabric and nodes power down sequence required as part of a full system power down may be initiated via the dashboard, the complimentary

full system power up sequence is initiated prior to the dashboard being available for use.

In the absence of an Application Manager, the dashboard is unable to show how an application maps to the system resources, nor provide any control over the deployment and management of applications.

6.8.1. Dependencies

- Monitoring Service
- Assembly Agent
- Librarian

6.8.2. Initial Architecture Plans

The Management Dashboard will run on the top of rack server and will integrate with a small number of system components.

6.8.3. Future Architecture Plans

Integration with other sources would extend the capabilities of the dashboard. For example, integration with an Application Manager would permit application oversight and control. Integration with the Authentication and Authorisation Services would enable user/tenant management and make it easy to understand what resources were being consumed by which user/tenant.

Utilisation of a future Monitoring Service publish/subscribe interface will make acquiring information for visualisation much more efficient. Encouraging other management software components to use the Monitoring Service for communicating state changes, e.g. Librarian Book allocations, as they happen would also be more efficient.

There is no specific need for the Management Dashboard to run within a single machine instance and could be run independently and integrate with multiple machine instances simultaneously to provide a multi-system view.

6.9. Application Manager

The application manager provides mechanisms for installing applications within The Machine.

6.9.1. Requirements

Stage application binaries within The Machine and deploy them to nodes as appropriate.

6.9.2. Initial Architecture

1. Each user within the machine will have SSH access to both nodes within the machine and the management server. As described in Section 6.4, “Login”, these will all share /home and /srv directories.

2. Users may stage information in /srv or /home as appropriate.

3. Users may use Ansible or custom shell scripts to install applications on nodes within the machine, either installing them to /home and /srv, or within the ram file system provided by the manifested node operating system.

4. MPI, SSH or other mechanisms may be used to start applications on each node.

6.9.3. Future Architecture

A web-based application deployment system will be run as a management service. This may use containers or other application packaging techniques to ease the process of application deployment to the nodes. Login access to the management server will not be required.

6.10. Installation and Updates

For Software Architecture Version 2, the only software installation and update system provided will be the standard Debian infrastructure. We’ll be looking at systems like Ansible and container-related technologies in the future to see what systems

6.11. Spill and Fill

The Spill/Fill system allows data to be moved to and from The Machine with relatively good performance. Given the amount of memory on The Machine users can expect saving data to take on the order of tens of minutes, depending on the amount of data being moved.

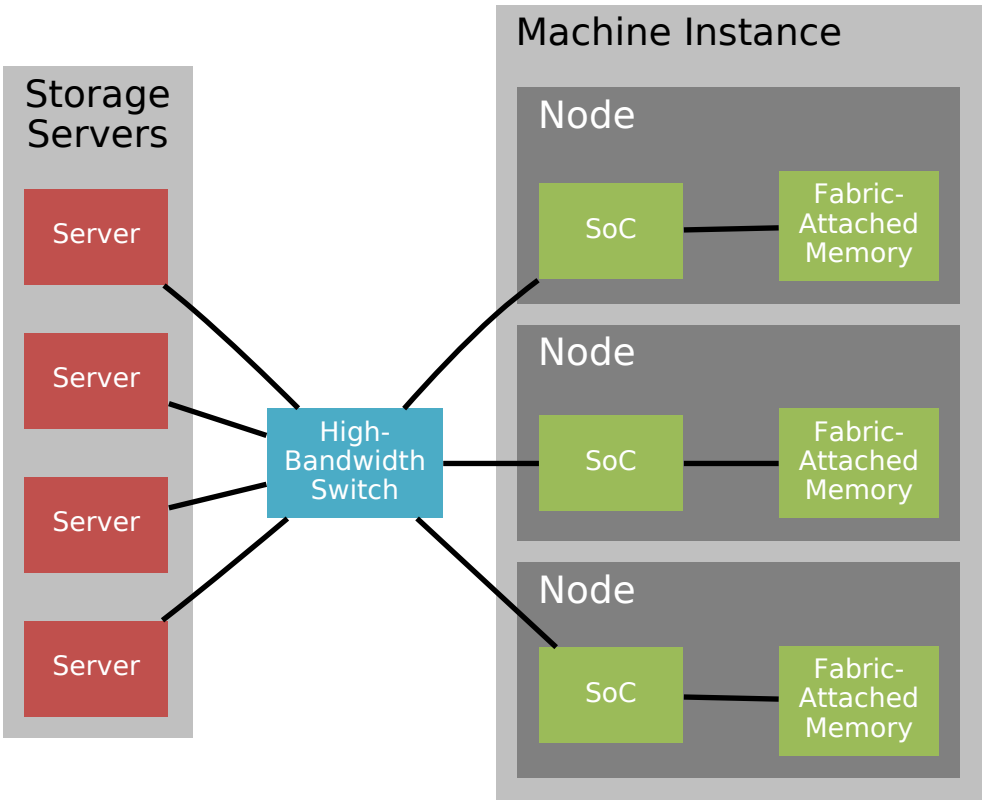
The system described in this section is scriptable so that we can build additional tools on top of it. It is not a complete product, but can be used by future tools that may become products.

The main goal of the spill/fill system is to provide an interface to transfer data between The Machine and a cluster of external servers, collectively known as the *Storage Cluster*, which contain non-volatile storage with sufficient performance to allow multiple users to share hardware over the course of a day.

The spill/fill interfaces provide users interfaces to see and manage the data stored in non-volatile storage. Users can see what datasets are stored in the spill/fill system and how much space the datasets occupy. Users can also see how much free space is available and delete datasets to make room for new datasets.

The hardware architecture for the spill/fill system consists the *Storage Cluster* connected, via a high-bandwidth network to one or more instances of The Machine as seen in Figure 6.3, “Spill/Fill Hardware Architecture”. The storage cluster for the spill/fill should be able to spill and fill from any instance of The Machine on the local network. Spill/fill to a remote instance of The Machine is not practical due to bandwidth limits between datacenters.

Figure 6.3. Spill/Fill Hardware Architecture



Spill/Fill is made available to the user as a regular command line application, *spillfill*, installed as part of the operating system on each of the nodes of The Machine.

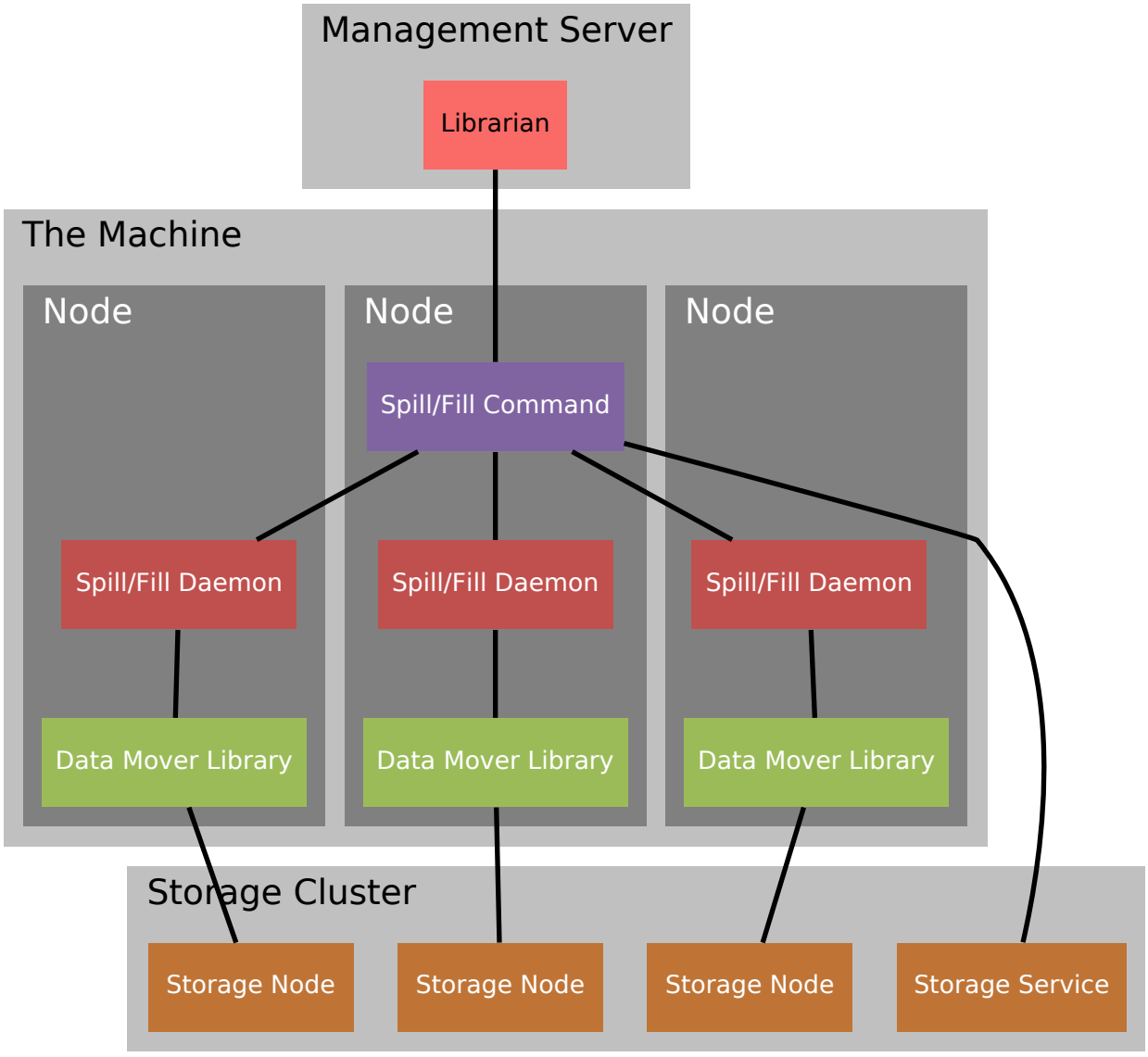
6.11.1. Theory of Operation

In the absence of performance criteria, an application as simple as *tar* or *rsync* would be entirely sufficient for saving and restoring data for the machine. However, operating across large volumes of data in a purely serial fashion from a single node could cause the operation to take hours or days.

To improve performance, each SoC in the system needs to be directed to operate on Fabric-Attached memory proximate to it. This will take advantage of both the parallel bandwidth available for non-conflicting operations within the fabric and the aggregate bandwidth of the individual SoC network connections.

The spill/fill system copies data to and from multiple SoCs in parallel between multiple nodes of the storage cluster. When data is spilled from an instance of The Machine to the storage cluster, the data is stored in a manner that allows it to be copied quickly and is not necessarily stored in files that neatly align with the shelves from which the data originated. An overview of this architecture can be seen in Figure 6.4, “Spill/Fill Software Architecture”.

Figure 6.4. Spill/Fill Software Architecture



6.11.2. Spill/fill application

Spillfill operates in a fashion similar to tar, either saving or restoring a list of files from a storage service. It also offers modes to manage the collection of datasets stored on the cluster by listing and deleting them.

Spillfill uses ssh to remotely execute *spillfilld* on the other nodes within The Machine. For those familiar with them, *spillfill* and *spillfilld* works much like rsync and scp and their associated daemons.

Name

spillfill — save and restore fabric attached memory

Synopsis

```
spillfill --list-sets
spillfill --delete-set [dataset ...]
spillfill --list [dataset ...]
spillfill --create [dataset] [pathname ...]
spillfill --extract [dataset] [pathname ...]
```

Description

Spillfill stores and extracts fabric attached memory files from a storage cluster.

One of the above operation modes must be specified to control what spillfill does.

Operation Modes

`--list-sets`

List all of the sets saved in the storage service.

`--delete-set` *[dataset ...]*

Deletes the named sets from the cluster.

`--list` *[dataset ...]*

List the contents of the specified datasets.

`--create` *[dataset] [pathname ...]*

Create a new dataset containing the specified contents.

`--extract`

Extract the specified contents from the specified dataset.

Other Options

`--exclude` *PATTERN*

exclude files that match the specified *PATTERN*, which is a shell glob expression.

`--exclude-ignore` *FILE*

exclude files whose names match patterns in *FILE*, one pattern per line.

`--verbose`

in --list mode, provide additional information about each file, including size, last modified date, owner/group values and xattrs relating to interleave groups.

`--server` *HOSTNAME*

This specifies the name of the host running the storage service. By default, spillfill uses the value configured in /etc/spill-fill.conf. If no host is configured or selected on the command line, spillfill exits with a suitable error message and error status.

`--port` *PORT*

This specifies the port on the host to connect to the storage service. The default value is 29805.

`--user` *USERNAME*

This specifies the username to present to the server in the connection setup. By default, the username associated with the current process is sent.

Files

`/etc/spill-fill.conf`

This file contains configuration parameters for *spillfill* in .ini file form. There is currently only one section *[service]* and two parameters — *hostname* which defines the default storage server, and *port* which defines the network port.

```
[service]
hostname=storage-server.mynet.mycorp.com
port=29805
```

6.11.3. Spill/fill daemon

Spillfilld is started via ssh by *spillfill*. Command line options direct it as to the mode of operation (create or extract) along with the storage node to communicate

with. The list of pathnames and ranges within those to operate on is received via standard input. Status messages are sent via standard output.

Each range within the shelf is sent to the storage server and the associated data sent or received as appropriate.

Name

spillfilld — daemon for saving and restoring fabric attached memory

Synopsis

```
spillfilld --server=[host name] --port=[port number]
```

Options

`--server` *host-name*

Specifies the hostname of the storage node to connect to.

`--port` *port-number*

Specifies the port number that the storage node daemon is listening on.

Description

Spillfilld transmits data between fabric attached memory shelves and the storage server. The server to contact must be specified on the command line.

Spillfilld processes send and receive messages read from standard input and sends status messages back on standard output. The format of these messages is described in Section 6.11.13, “*Spillfill* / *Spillfilld* communications”

6.11.4. Storage service

Storaged is run on one of the machines in the storage cluster. It manages the datasets, tracking dataset ownership, all of the shelf names and where each book of each shelf in the dataset is stored.

It is started at boot time as a system daemon so that it will be available at any time.

Name

storaged — storage cluster control daemon

Synopsis

```
storaged
```

Description

Storaged accepts inbound TCP/IP connections from *spillfill* and then processes messages on those connections. It is responsible for managing the storage nodes within the cluster, tracking how much space each node has available for dataset storage and the metadata and book locations for every shelf in the dataset.

Configuration file

Storaged needs a few configuration parameters, all of which will be stored in an .ini file called `/etc/storaged.conf`. This contains a [network] section which describes the connection setup and then a list of [node_xxx] sections which describes the cluster. Each node_xxx section describes one of the nodes in the cluster.

```
[network]
port=29805

[node_1]
id=1
hostname=host1.cluster.mydomain
port=21581
```

6.11.5. Spill/Fill Storage Node service

storenoded runs on all of the machines in the storage cluster. It must keep a database of where in the storage array each saved book is located so that the fill process can pull that data out as requested.

It is started at boot time as a system daemon so that it will be available at any time.

Name

storenoded — storage node daemon

Synopsis

```
storenoded
```

Description

Storenoded accepts inbound TCP/IP connections from *spillfilld* and then processes *Spill* and *Fill* messages from it. It is responsible for recording where each saved book is stored so that it can retrieve it when requested.

Configuration file

Storenoded needs a single configuration parameter — the port to listen on. That is contained in the [network] section of the /etc/storenoded.conf file.

```
[network]
port=21581
```

6.11.6. Create a new Dataset

For `spillfill --create [set] [pathnames]`, *spillfill* will:

1. Ask the storage server if the destination dataset already exists, exiting with an error message if it does.
2. Generate the full set of shelves to save and their sizes.
3. Compute the total space required within the storage server and verify with the storage server that sufficient space is available.
4. Discover the interleave group where each book within each shelf is stored.
5. Find the mapping between interleave groups and the network name of a suitable a near-by SoC.
6. Query the storage server for the network names of the available storage nodes, and the amount of storage available on each one.
7. Construct a mapping between interleave groups and storage nodes such that each interleave group fits in a single storage node. If no such mapping can be constructed because there is not sufficient space on the nodes, then exit with an error message.
8. Start *spillfilld* processes on each of the necessary SoCs using ssh. Specify on the command line to each process the destination storage node network name.
9. Send the appropriate list of shelf names and file offsets to each *spillfilld* process.
10. Collect status information back from each *spillfilld* process. Report errors to the user. If an error does occur, delete the failed dataset from the storage server. Exit with zero status when successful, otherwise exit with a non-zero status.

6.11.7. Extract a Dataset from the server

For `spillfill --extract [set] [pathnames]`, *spillfill* will:

1. Ask the storage server if the source dataset exists, exiting with an error message if it does not.
2. Get information from the storage server about all of the shelves in the dataset:

   a. name

   b. size

c. owner, group

d. mode

e. xattrs specifying both the requested and actual interleave groups for each book within the shelf.

f. Storage node containing each book within the shelf.

3. Create all of the target shelves with the correct sizes, requested allocation parameters, ownership and mode. If any of the target shelves already exists, exit with an error status, cleaning up by removing any shelves that were created.

4. Construct a mapping between storage nodes and the actual interleave groups where shelves are stored. Select suitable SoCs for each storage node based on the target interleave groups for books coming from the node.

5. Start *spillfilld* process on each of the necessary SoCs using ssh. Specify on the command line to each process the source storage node network name.

6. Send the appropriate list of shelf names and file offsets to each *spillfilld* process.

7. Collect status information back from each *spillfilld* process. Report errors to the user. If an error does occur, clean up by removing all created shelves. Exit with a zero status when successful, otherwise exit with a non-zero status.

6.11.8. List files in a dataset on the server

For `spillfill --list [set]`, *spillfill* will:

1. Ask the storage server if the source dataset exists, exiting with an error message if it does not.

2. Get information from the storage server about all of the shelves in the dataset:

a. name

b. size

c. owner, group

d. mode

e. xattrs specifying both the requested and actual interleave groups for each book within the shelf.

f. Storage node containing each book within the shelf.

3. Display this information to the user. If `--verbose` is specified on the command line, then all of the information will be displayed. Otherwise, only the name of the shelf will be displayed.

6.11.9. List datasets on the server

For `spillfill --list-sets`, *spillfill* will:

1. Ask the storage server for a list of all available datasets, along with information about their size
2. Sort the list by dataset name
3. Present the full list to the user, with one line per dataset.

6.11.10. Delete datasets from the server

For `spillfill --delete-set [sets]`, *spillfill* will:

1. Ask the storage server for a list of all available datasets
2. Match the available sets with the patterns supplied on the command line. It is an error to match no sets.
3. Send the selected set to delete to the storage server.
4. Receive status information and report back to the user.

6.11.11. Message contents

Each message is a data structure consisting of numbers, strings, lists, bulk-data and nested data structures as specified in Table 6.2, “Spill/Fill Types”

Table 6.2. Spill/Fill Types

Type	Contents
number	64-bit signed integer
enum	Enumerated type
string	String of arbitrary contents and length
bulk-data	array of raw bytes
list	list of values of the same type
structure	Aggregate of numbers, strings and structures

In this document, message structures will be described like:

allocation

```
┌──
    storage-node        number
    start-offset        number
    end-offset          number
└──
```

shelfinfo

```
┌──
    name                string
    size                number
    allocations         list of allocation
└──
```

A dialog between two processes over the network will be described in a unified format, with the communication in one direction separated from the communication in the other direction with "▶":

```
┌──
    List-Datasets       message-id
    sequence            number
    dataset             string
  ▶
    Datasets            message-id
    sequence            number
    status              number
└──
```

The precise encoding of the communication will be determined by the team implementing the system. Something like json might work fine.

Message-id is an enumeration of all possible message identifiers in both directions. An error can be returned at any point; the sequence value identifies the triggering message.

Status Values

Status is returned by several commands and is one of the following:

Status	**Meaning**
Success	The operation completed normally

Status	Meaning
NoPermission	The user hasn’t the necessary permissions
NoSuchSet	The named dataset does not exist
NoSpace	There is insufficient space.
AlreadyExists	The named dataset or shelf already exists

Common Datatypes

Here are a few compound datatypes shared between multiple messages:

Attribute

```
┌──
      name                  string
      value                 string
└──
```

Stores additional attributes about the file in a free-form property list. Mostly useful for extended attributes.

Extent

```
┌──
      starting-offset       number
      ending-offset         number
└──
```

A region of a shelf which extends from starting-offset through the byte just before ending-offset; the number of bytes in the extent is ending-offset - starting-offset.

SavedBook

```
┌──
      node-id               number
      starting-offset       number
      ending-offset         number
└──
```

Information about a single book saved in the dataset, including which node it is stored on and the extent of the shelf recorded in it. See The Extent Datatype for a description of the offset values.

ShelfInfo

name	string
shelf-id	number
size	number
uid	number
gid	number
mode	number
attributes	list of Attribute
books	list of SavedBook

Information about a shelf saved in the dataset, including all of the necessary attributes and the list of saved books.

6.11.12. *Spillfill / Storaged* communications

These two programs communicate over a network socket opened by *spillfill*. *Spillfill* sends the storage server requests for information about the contents of the storage server using these messages:

Set User

Set-User	message-id
sequence	number
username	string
▶	

Set the username of the associated connection. See Section 6.11.16, “Security” for a description of how this is used. No reply is generated.

Check Dataset

Check-Dataset	message-id
sequence	number
dataset	string
▶	
Dataset-Status	message-id

sequence	number
status	Status

If *dataset* exists, return Success, else return NoSuchSet.

List Datasets

List-Datasets	message-id
sequence	number

▶

Dataset-Names	message-id
sequence	number
datasets	list of string

The list of datasets owned by the current user on the storage server will be returned.

List Nodes

Node

id	number
hostname	string
port	number
total-space	number
free-space	number

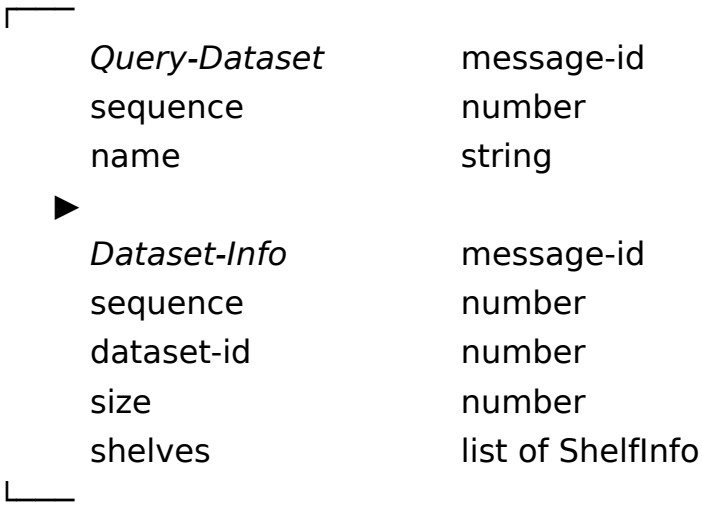
List-Nodes	message-id
sequence	number

▶

Nodes	message-id
sequence	number
datasets	list of Node

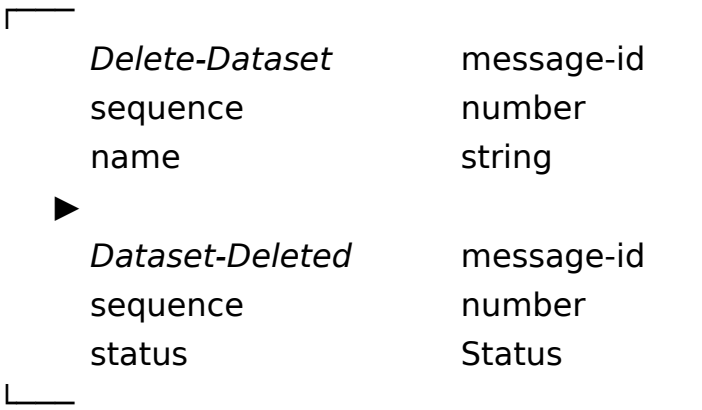
Lists the storage nodes in the cluster, providing information about how to access them and both the total and free space amounts (in bytes) on the node.

Query Dataset



Returns all of the information about a dataset and the shelves contained therein.

Delete Dataset



Attempts to deletes the named dataset, returning Success if it was deleted or an error otherwise.

Create Dataset

name	string

▶

Dataset-Deleted	message-id
sequence	number
dataset-id	number
status	Status

└──

Creates the named dataset and return the id in dataset-id. If the named dataset already exists, returns AlreadyExists in status.

Add Shelf

┌──

Add-Shelf	message-id
sequence	number
dataset-id	number
shelf-info	Shelf-Info

▶

Shelf-Added	message-id
sequence	number
status	Status

└──

Adds a shelf to the specified dataset. dataset. Reports an error if the shelf name or shelf id is already in the dataset, or if there is not enough space in one or more of the nodes listed in the book information for the shelf.

6.11.13. *Spillfill / Spillfilld* communications

These two communicate over the socket set up by the ssh command that starts *spillfilld*. Data sent to *spillfilld* will be received on standard input, while data sent via standard output will be delivered back to the controlling *spillfill* process.

Send

┌──

Send	message-id
sequence	number
dataset-id	number

name	string
shelf-id	number
extents	list of Extent

▶

Sent	message-id
sequence	number
status	Status

└──

Instructs *spillfilld* to open the named shelf and deliver the specified extents of it to the storage node.

Receive

┌──

Receive	message-id
sequence	number
dataset-id	number
name	string
shelf-id	number
extents	list of Extent

▶

Received	message-id
sequence	number
status	Status

└──

Instructs *spillfilld* to open the named shelf, fetch the specified extents for it from the storage node and write them into the shelf.

Done

┌──

Done	message-id
sequence	number

▶

Done	message-id
sequence	number

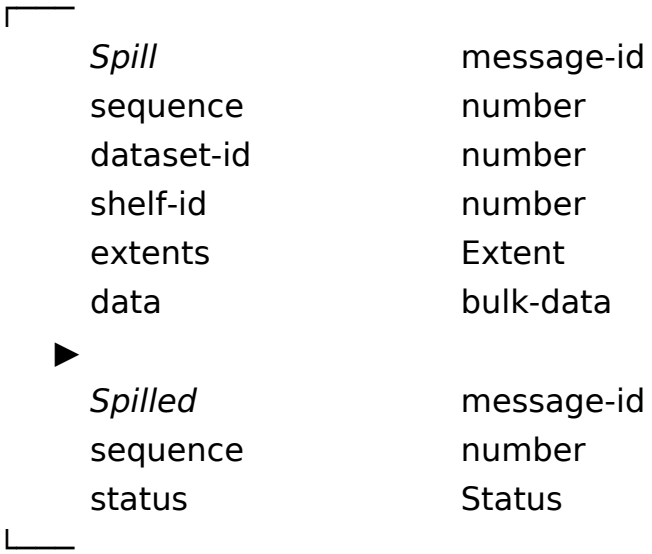
└──

Provides a synchronization point so that *spillfill* can tell when *spillfilld* is finished. Upon receipt of this message, *spillfilld* will first send the response back and then close the network connection and terminate the process.

6.11.14. *Spillfilld / Storagenoded* communications

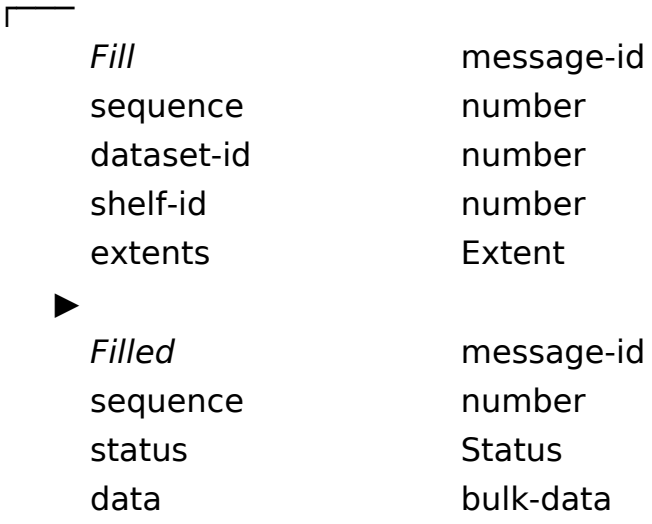
Spillfilld connects to *storagenoded* using the server and port parameters passed to it from *spillfill*.

Spill



Sends data from the shelf to be stored in the storage node.

Fill



└──

Retrieve saved data from the storage node to be written to the shelf.

6.11.15. Data Mover Library

The Data Mover Library is a set of helper functions to accelerate copying of data from Fabric Attached Memory to the network. It is used by *spillfilld* to improve data throughput, and is otherwise fairly transparent to the operation of the Spill/Fill system.

6.11.16. Security

The current design tags each dataset with the Linux username associated with *spillfill* which created the dataset. This provides some protection against destroying data belonging to other users, but is not in any way secure. In a future addition, the spill/fill system will want to control access to the storage service, allowing each owner to create, list, extract and delete only their own datasets.

In the current design, all network communications between processes is unencrypted, allowing visibility of application data on the network link between the two ends. In a future addition, this data will be encrypted.

6.12. Time Services

The Data network will have a PTP appliance present. The management server and the node operating systems will all run ptpd to ensure that they’re clocks are synchronized.

The Management network does not have access to the PTP information, so the management server will run ntpd to provide time over NTP to systems on the management network.

6.12.1. PTP

The PTP Daemon (https://github.com/ptpd) is the upstream source for the Debian (and hence TMD) ptpd package. It uses kernel interfaces to access capabilities in the underlying MAC or PHY hardware to closely synchronize clocks across the network.

For robustness, the management server will run as a PTP master device (ptpd option -m), using the grandmaster device when that is available and failing over to provide PTP services to keep all of the nodes in the machine sychronized together even when the grandmaster clock is not available.

Nodes will run as slave devices (ptp option -s) to avoid changing the controlling clock as they boot at shut down.

6.12.2. NTP

Within the management network, NTP is used to provide a vague notion of time to the management processors. Lacking a dedicated NTP server, the management server will fill that role.

The ntpd on the management server will offer time data but to never adjust the lock clock as that is controlled by PTP. That is done by using only the local clock

```
# /etc/ntp.conf
server 127.127.1.0 prefer
fudge  127.127.1.0 stratum 5
```

Chapter 7. Provisioning

Provisioning the machine includes processes for manifesting root file systems, booting operating system images to bare metal nodes, managing virtual machines within the machine, along with rudimentary user login access.

7.1. Initial Booting

Each node will be configured to boot over the network using PXE. The other boot option available is over USB, and while that may prove a useful tool during initial hardware bring-up, we will need to be able to manage the machine configuration from a single point.

PXE (i.e., DHCP/TFTP/BOOTP) will be served by the Top of Rack Management Server. An initial kernel and ramdisk image can be customized per node depending on tenant and demo/workload configuration. The file system image provides, at a minimum, sufficient L4TM packages and configuration thereof for the node to join the overall cluster management system:

- complete the network boot and start necessary daemons
- connect to the local intranet
- connect to the global Librarian File System
- allow remote logins via LDAP authentication to ToRMS
- perform admin tasks as a remote client via ssh/ansible
- (optional) NFS mount auxiliary data for demos

7.2. Resource Discovery

Each node within the machine will have a set of SoC and FAM resources which must be known to the cluster at large. In particular, the Assembly Agent and Librarian need to know the physical topology of The Machine.

7.2.1. Initial Implementation Plan

The set of nodes, their SoC and FAM resources and identity information will be statically configured within the Top of Rack Management Server. This will be driven by a hand-crafted configuration file which completely describes the topology of The Machine in use. Refer to Chapter 5, *Configuration Data* for details.

7.2.2. Eventual Implementation Plan

The nodes should announce their presence within the cluster and dynamically provide all necessary configuration information, including SoC and FAM resources. Lots of additional verification and validation will be performed to make sure the node can be integrated into the system.

7.3. OS and Root FS Instantiation

The choice of OS image per node will be based on the network boot of custom kernels and file system images. Classic "provisioning" will not apply (i.e., installation to local mass storage followed by a reboot from that storage).

7.3.1. Requirements

Manifest an L4TM image for each node within the cluster, and configure a PXE boot server to deliver the appropriate images to each node.

7.3.2. Initial Implementation

As there is no practical local per-node storage, nodes will boot in a diskless fashion over The Machine intranet. A kernel image and initial RAM disk will be delivered by the PXE server. Conventional diskless configurations (such as LTSP) use NFS to realize the final root file system. For The Machine we want to avoid undue strain on the intranet, i.e., 80 busy NFS mounts.

Thus each "initial ramdisk" is actually a full OS image with L4TM and possibly demo/workload packages. It will run completely from RAM, obviating the need for network access for typical file system operation.

The goal is to keep the image under 1% of RAM (2.5 Gb). L4TM will take about .5 Gb, leaving 2 Gb for demo packages and temporary storage. Should a particular demo require more space, then NFS can be used for secondary storage.

The manifesting process centers on the Debian "vmdebootstrap" command. It’s a smart wrapper around the legacy debootstrap command, intended to create a hard disk image for virtual machines. With a little customization the resulting "pristine" image becomes a per-node root file system image. This image is loaded over the network into RAM, and the node’s root file system is mounted from that RAM image. Such an image contains

- Base L4TM packages (driven by debootstrap)
- Additional Debian-derived L4TM packages for The Machine (system and network utilities)

- Custom HP packages from L4TM (Librarian, atomic library)
- Node identification (name, IP address, etc) and initialization
- Basic user configuration to support LDAP-referenced login
- "ansible" configuration to act as a remote client

Other customization can be made to support demos (packages, raw files, config scripts and the like).

7.3.3. Second implementation

Once FAM is stable enough to hold a file system, we'll change the provisioning service to create FAM regions large enough for root file systems for each node, and install the system to that. A proof point could be to use FAM as temporary storage (/tmp).

Chapter 8. OS Manifesting

Getting a specific instance of Linux running on set of nodes of The Machine involves a sequence of steps:

1. Selecting the desired packages and other configuration options.
2. Constructing the kernel and root file system for each node.
3. Installing the resulting bits in the PXE system
4. Configuring PXE to supply the appropriate bits to the desired nodes.
5. Directing the desired nodes to boot.

The first four steps of this process are what we refer to as “Manifesting” and are performed by the OS Manifesting Service, which offers both network- and command-line- based interfaces.

The fifth step is separate. Once manifesting is complete and a file system and kernel are prepared, then the assembly agent will communicate with the node management processors and direct them to reboot the SoC.

8.1. OS Manifest Specification

A “manifest” is a specification for how to construct the OS and Root FS from the distribution repository. It contains both package information and details on how to customize that set of packages for the particular node. As with much of The Machine, we’ll be using JSON to contain this information.

The manifest has a header containing a description of the manifest and various global options along with the set of packages to install from the repository.

As a convenient shorthand, the manifest may specify a list of “tasks”. Each task defines a selection of packages for a broadly defined task and comes from the Debian tasksel system.

The top level structure looks like:

```
{
        "_comment": "Comments are allowed at any point",
        "name": "manifest name",
        "description": "a description of the contents and intended use",
        "release": "the operating system release (unstable/testing/stable)",
        "tasks" :
        [
                …
        ],
        "packages" :
        [
```

```
                …
        ]
}
```

Table 8.1. Manifest Objects

Name	Required	Contents
name	yes	Name of manifest
description	no	Details about manifest
release	no	Target OS release name
tasks	yes	Array of tasks
packages	yes	Array of packages

8.1.1. Tasks

Tasks represent collections of packages designed to provide a specific functionality. Here's a sample of the tasks that will be available.

Table 8.2. Available Tasks

Name	Required	Description
java	No	Java runtime
c-dev	No	C/C++ development
java-dev	No	Java development

The JSON encoding of the list of tasks is an array of the task names desired.

```
"tasks":
[
        "java"
]
```

8.1.2. Packages

Each package to be installed can also specify the release or package version to more closely specify which of a variety of available packages to install. All dependencies of the package will also be installed. Each package may also specify whether to install recommended or suggested packages as well.

```
"packages":
[
        {
                "package": "nickle",
                "version": "2.77-1",
                "release: "unstable",
                "install_recommends": "true",
                "install_suggests": "true"
        }
```

]

Table 8.3. Package Objects

Name	Required	Contents
package	yes	name of package
version	no	version specification
release	no	OS release name
install_recommends	no	Defaults to "yes"
install_suggests	no	Defaults to "no"

By default, the global “release” object will control which release to pull package from, but each package may override that. Note that using multiple release names can easily cause conflicts that will make it impossible to construct a valid result.

8.2. Manifesting Details

The file system constructed for each node contains the installed versions of the desired packages along with some data specific to the node and machine instance involved:

Configuration Data

As described in Chapter 5, *Configuration Data*, the configuration data for the entire machine is to be made available in the node operating system as /etc/tm-config

TLS Private Certificate

The TLS public certificate for all nodes and services is available in the configuration data file, but the private certificate for each node must be known only to itself. Ideally, this would be something stored in the node itself, but until then the TLS private certificate for each node will be stored in /etc/ssl/private/ssl-cert.key. The public certificate will be stored in /etc/ssl/certs/ssl-cert.pem

NFS Configuration

To provide some stable storage for users of The Machine, /home and /srv directories from the management server will be made available via the same name on each node using NFS

LDAP Configuration

Login data for user accounts on the nodes will come from the central LDAP database. The LDAP service on the nodes must be configured during manifesting to use the LDAP service running on the management server.

/etc/hosts file

To avoid needing a functioning DNS environment, each node will be provided with an /etc/hosts file which provides names and addresses for all of the nodes, the management server and the spill/fill server (if present in the environment).

/etc/hostname

This provides the hostname for the node as seen in the /etc/hosts file.

Node Coordinate

The node coordinate will be fetched from ACPI on TMAS or hardware. For FAME, the node coordinate will be placed in /etc/tm-coordinate. Applications needing the current node coordinate will need to first check ACPI, and if that fails, fall back to /etc/tm-coordinate.

8.3. OS Manifesting Service and API

The Machine OS Manifesting Service (TMMS) runs on the Management Server and provides a way to manage manifests and direct the configuration of the PXE service.

The OS Manifesting Service runs as an HTTP server that accepts HTTPS connections to receive requests and perform operations via a REST interface. This outward interface has the following qualities:

- HTTPS 1.1 (TLS)
- Representation State Transfer (REST)
- JavaScript Object Notation (JSON) is used to exchange models and state
- UTF-8 encoding
- The current version of this protocol is 1.0

To allow for changes in the procotol, the client should transmit the version it is using in the request’s Accept header:

- Accept: application/json; version=1.0

Based on the above qualities, every PUT, POST and GET request should have the following HTTP header, which describes the format of the data contained in the body of the request:

- Content-Type: application/json; charset=utf-8; version=1.0

The interface supports four HTTP methods: GET, PUT, POST and DELETE. This document describes the external API of the TMMS, using these methods. Unless

otherwise stated, the URLs supported by the TMMS will work with or without a trailing slash (“/”); for example, both the URI “/manifest” and “/manifest/”are equivalent.

Although the methods described in this document show the possibility of the HTTP response code “401 Unauthorized”, the TMMS does not yet support authorization; the intent is to add support for authorization in a future release.

8.3.1. OS Manifesting Data Storage

OS Manifesting data is stored in the management server file system in /var/lib/tmms. It may also place cached intermediate manifesting data in /var/cache/tmms.

/var/lib/tmms

Root of the manifesting services information

/var/lib/tmms/manifest

Directory containing manifests.

/var/lib/tmms/node

Directory containing node manifest associations.

/var/lib/tmms/node/machine_rev%2f1%2fdatacenter%2fpa1%2fframe%2fA1.above_floor%2frack%2f1%2fenclosure%2f1%2fnode%2f1

File containing the manifest path name for the node at coordinate machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/enclosure/1/node/1 in the form:

```
{
        "manifest": "manifest1.json"
}
```

/var/lib/tftpboot

Root of TFTP file system

/var/lib/tftpboot/pxelinux.cfg

Directory of PXE configuration files

machine_rev%2f1%2fdatacenter%2fpa1%2f /var/lib/tftpboot/pxelinux.cfg/7c-7a-91-24-c8-3b:: PXE configuration for node with MAC address 7c:7a:91:24:c8:3b. Note that nodes will have multiple NICs, so the PXE configuration must be available at all addresses. This can be done by hard linking multiple names to the same file. This file will contain:

```
DEFAULT tmd/machine_rev%2f1%2fdatacenter%2fpa1%2fframe%2fA1.above_floor%2frack%2f1%2fenclosure%2f1%2f
```

```
INITRD tmd/machine_rev%2f1%2fdatacenter%2fpa1%2fframe%2fA1.above_floor%2frack%2f1%2fenclosure%2f1%2fn
```

/var/lib/tftpboot/tmd

Root of tree holding kernels and file systems for each node

/var/lib/tftpboot/tmd/machine_rev%2f1%2fdatacenter%2fpa1%2fframe%2fA1.above_floor%2frack%2f1%2fenclosure%2f1%2fnode%2f1

Directory holding kernel and file system for node at coordinate machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/enclosure/1/node/1.

/var/lib/tftpboot/tmd/machine_rev%2f1%2fdatacenter%2fpa1%2fframe%2fA1.above_floor%2frack%2f1%2fenclosure%2f1%2fnode%2f1/vmlinuz

Linux kernel for node at coordinate machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/enclosure/1/node/1.

/var/lib/tftpboot/tmd/machine_rev%2f1%2fdatacenter%2fpa1%2fframe%2fA1.above_floor%2frack%2f1%2fenclosure%2f1%2fnode%2f1/initrd.gz

Root file system for node at coordinate machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/enclosure/1/node/1.

8.3.2. OS Manifesting Data Persistence

To make the overall system more resilient to transient failure of the OS Manifesting Service (TMMS) and other management infrastructure, the data managed by the service will persist across service instances.

At startup time, TMMS will use the existing configuration found in the file system as specified in Section 8.3.1, “OS Manifesting Data Storage”. During execution, TMMS will update that configuration in response to API calls.

Any temporary data which may have been stored by a previous instance of TMMS in /var/cache/tmms will be removed at TMMS service startup and during a clean shutdown of the service.

8.3.3. Listing available tasks and packages

List Available Packages

Description

Download the list of all packages available.

Resource

/packages

HTTP Method

GET

Response

200 OK: The list of packages is provided in a JSON structure of the form:

```
{
        "packages": [
                {
                        "package": "nickle",
                        "version": "2.77-1+b1",
                        "description": "desk calculator language"
                },
                {
                        "package": "altos",
                        "version": "1.6.2-1",
                        "description": "Altus Metrum firmware and utilities"
                }
        ]
}
```

Table 8.4. Package List Details

Name	Contents
package	name of package
version	version specification
description	package description

Query Available Package

Description

Get details about a specific package

Resource

/package/{name}

HTTP Method

GET

Input

name: the name of the package

Response

200 OK: Information about the package is provided as a JSON structure of the form:

```
{
        "package": "nickle",
        "version": "2.77-1",
        "installed-size": 1027,
        "maintainer": "Keith Packard <keithp@keithp.com>",
        "architecture": "amd64",
        "depends":
        [
                {
                        "package": "libc",
                        "version": ">= 2.3.4"
                },
                {
                        "package": "libreadline",
                        "version": ">= 6.0",
                },
                {
                        "package": "libtinfo5",
                },
        ],
        "description-en":
        [
                "desk calculator language",
                "Nickle is a language with powerful programming and scripting capabilities."
                "Nickle supports a variety of datatypes, especially arbitrary precision"
                "integers, rationals, and imprecise reals.  The input language vaguely"
                "resembles C.  Some things in C which do not translate easily are different,"
                "some design choices have been made differently, and a very few features are"
                "simply missing."
        ],
        "description-md5": "e74e3f8e1a10529c636a806fe87cb981",
        "tag": [
                "devel::interpreter",
                "field::mathematics",
                "interface::text-mode",
                "role::program",
                "scope::utility",
                "uitoolkit::ncurses"
        ],
        "section": "interpreters",
        "priority": "optional",
        "filename": "pool/main/n/nickle/nickle_2.77-1_amd64.deb",
        "size": 590682,
        "md5sum": "3e71799db516ebe944facf28e900ea27",
        "sha1": "990ad55dc9239a2c8d1fe44e2b3506211dabf380",
        "sha256": "9231e57a2e2e3639d240c35c452bce4dafd546b4c86daa333f98f7558adede2f"
}
```


Table 8.5. Package Details

Name	Contents
package	name of package
version	version specification
installed-size	space required in kiB
maintainer	contact information for maintainer
architecture	package architecture

Name	Contents
depends	list of dependencies
description-en	package description in English
description-md5	md5 sum of description
tag	list of package tags
section	archive section
priority	package priority
filename	filename in archive
size	binary package size in bytes
md5sum	MD5 checksum of package file
sha1	SHA1 checksum of package file
sha256	SHA256 checksum of package file

List Available Tasks

Description

Download the list of all tasks present in the service.

Resource

/tasks

HTTP Method

GET

Response

200 OK: The list of tasks available

```
{
        "tasks": [
                "java",
                "c-dev",
                "java-dev"
        ]
}
```

Query Available Task

Description

Download the list of packages defined by a task.

Resource

/task/{name}

HTTP Method

GET

Input

name is the name of the task to query.

Response

200 OK: The package names defined by the specified task is provided as a JSON structure of the form:

```
{
        "packages": [
                "nickle",
                "altos",
                …
        ]
}
```

8.3.4. Managing Manifests

The Machine OS Manifesting Service (TMMS) stores a list of manifests that can be used to configure the OS for a set of nodes. Those manifests live in the */manifest* tree of the manifesting service. Manifests may be uploaded and downloaded from the TMMS and the list of available manifests queried.

Upload Manifest

Description

Upload a new manifest to the service. The manifest may already exist, if so, it's contents are replaced by the new contents.

Resource

/manifest/{prefix}

HTTP Method

POST

Input

prefix: (optional) Manifest name prefix. This value is prepended to the name in the manifest itself to construct the full name of the manifest for other operations. This name may contain slashes.

Body

The body of the POST request consists of the manifest as described in Section 8.1, “OS Manifest Specification”. This includes the name of the manifest, which is appended to the *prefix* value to build the full manifest name.

Response

200 OK: An existing manifest has been replaced with the provided contents.

201 Created: The manifest has been created with the provided contents.

Download Manifest

Description

Download an existing manifest from the service.

Resource

/manifest/{manifest}

HTTP Method

GET

Input

manifest: (Required) Full manifest name. This is a complete manifest name, including the *prefix* value supplied when the manifest was created.

Response

200 OK: The body of the GET request consists of the manifest as described in Section 8.1, “OS Manifest Specification”.

Errors

404 Not Found: The specified manifest does not exist.

Delete Manifest

Description

Deletes an existing manifest from the service. Note that this simply deletes the manifest itself and that any nodes configured to use the manifest will continue to boot using the constructed kernel and root file system.

Resource

/manifest/{manifest}

HTTP Method

DELETE

Input

manifest: (Required) Full manifest name. This is a complete manifest name, including the *prefix* value supplied when the manifest was created.

Response

204 No Content: The specified manifest has been deleted.

Errors

404 Not Found: The specified manifest does not exist.

List Manifests

Description

Download the list of all manifests present in the service.

Resource

/manifest/{prefix}

HTTP Method

GET

Input

prefix: (Optional) This limits the returned manifests to those whose full name starts with the specified prefix. A complete list of all manifests is returned when *prefix* is empty.

Response

200 OK: The list of manifests whose name starts with the specified prefix is provided as a JSON structure of the form:

```
{
        "manifests": [
                "prefix-1/manifest1.json",
                "prefix-2/manifest2.json",
                …
        ]
}
```

204 No Content: No Manifests are available under the provided path

8.3.5. Configuring Nodes

Each node within the system has a manifest associated with it. When rebooted, the node will load an OS image constructed for it from the specified manifest. All node information is stored under the coordinate for the respective node.

Setting Node Manifest

Description

Specify the manifest for a node

Resource

/node/{node}

HTTP Method

PUT

Input

node: (Required) Node coordinate.

Body

manifest: (Required) The desired full manifest name is provided wrapped in a JSON structure of the form:

```
{
        "manifest": "pats_manifests/manifest1.json"
}
```

Response

200 OK: The manifest for the specified node has been changed. This means the build process for a fresh filesystem image has been started.

201 Created: The manifest for the specified node has been set. This means the build process for a fresh filesystem image has been started.

Error

404 Not Found: The specified node or manifest does not exist.

Getting A Single Node Manifest

Description

Query the manifest currently specified for a node

Resource

/node/{node}

HTTP Method

GET

Input

node: (Required) Node coordinate

Response

200 OK: The manifest path name for the specified node is returned in a JSON structure of the form:

```
{
        "manifest": "prefix-1/manifest1.json",
        "status": "ready",
        "message": "life is good"
}
```

204 No Content: There is no manifest associated with the specified node.

Error

404 Not Found: The specified node does not exist.

Table 8.6. Manifest Status Values

Name	Description
ready	Manifest is installed and ready for node to boot
building	Manifest building is in process
error	Manifest is not ready, see message for details

The “building” status is transient and will transition to either “ready” or “error”, depending on whether the building process succeeds or not. If the status is “error”, then the message value will contain additional information about the cause of the error which can be presented to the user. Once the list of error causes has been determined, we’ll list those here.

Getting All Node Manifests

Description

Query the manifests currently specified for all nodes.

Resource

/node

HTTP Method

GET

Response

200 OK: The manifests for all of the nodes is returned in a JSON structure of the form:

```
{
        "mappings" :
        {
                "frame/A1.above_floor/rack/1/enclosure/1/node/1":
                {
                        "manifest": "prefix-1/manifest1.json",
                        "status": "ready",
                        "message": "life is good"
                },
                …
        }
}
```

204 No Content: There are no manifests associated with any nodes.

Delete Node Manifest Association

Description

Remove the node configuration from the manifesting service. When the node is rebooted, there will not be an operating system or root file system made available to it. Future GET requests for this node will return an empty reply (*204 No Content*)

Resource

/node/{node}

HTTP Method

DELETE

Input

node: (Required) Node coordinate

Response

204 No Content: The manifest association for the specified node will be deleted.

Error

404 Not Found: The specified node does not exist.

Listing Nodes

Description

Query the list of available nodes

Resource

/nodes

HTTP Method

GET

Response

200 OK: The list of available nodes is returned in a JSON structure of the form:

```
{
        "nodes": [
                "frame/A1.above_floor/rack/1/enclosure/1/node/1",
                "frame/A1.above_floor/rack/1/enclosure/1/node/2",
                …
        ]
}
```

8.3.6. Protocol Security

Securing the OS Manifesting Service will happen at a later date, for now the system will operate without any checks.

In the future, the OS Manifesting service will use Keystone authorization tokens for security. Correct authorization is required to change the state of the OS manifesting environment:

- Upload Manifest
- Delete Manifest
- Setting Node Manifest

8.4. OS Manifesting Command Line Tools

For testing and debugging of the manifesting system, a command line tool, *tm-manifest*, which communicates with the OS Manifesting service is included.

8.4.1. Name

tm-manifest — manifesting for the machine

8.4.2. Synopsis

```
tm-manifest listpkgs
tm-manifest showpkg <package>
tm-manifest listtasks
tm-manifest showtask <taskname>
tm-manifest put <manifest name> <manifest file>
tm-manifest get <manifest name> <manifest file>
tm-manifest delete <manifest name>
tm-manifest list <manifest name>
tm-manifest listnodes
tm-manifest setnode <node-coordinate> <manifest name>
tm-manifest getnode <node-coordinate>
```

8.4.3. Options

`--server` *host-name*

Specifies the hostname of the management server to connect to.

`--port` *port-number*

Specifies the TCP port that the management server is listening on.

8.4.4. Description

Tm-manifest communicates with the OS manifesting service on the specified machine to set up manifests for nodes within the machine. It offers a selection of sub-commands for this purpose:

listpkgs

List available packages in the same format as *apt list*.

showpkg

Show details about the specified package in the same format as *apt show*.

listtasks

List the available tasks, one per line.

showtask

List the packages contained within the specified task, one per line.

put

Upload a manifest to the server with the specified name from the specified file.

get

Download a manifest from the server with the specified name to the specified file.

delete

Delete the specified manifest

list

Show the list of available manifests, one name per line.

listnodes

Show the coordinates of every node in the machine instance, one coordinate per line.

setnode

Select the manifest for the specified node and construct a kernel and root FS that the node will use the next time it boots.

getnode

Return the manifest name that the specified node is currently directed to use at next boot.

questions: Startup state Restart state Sharing information with the aa Construct a more formal task list with details list manifests shouldn’t take a manifest Declare lack of access control specification Defined manifests delivered with the system

Chapter 9. Storage

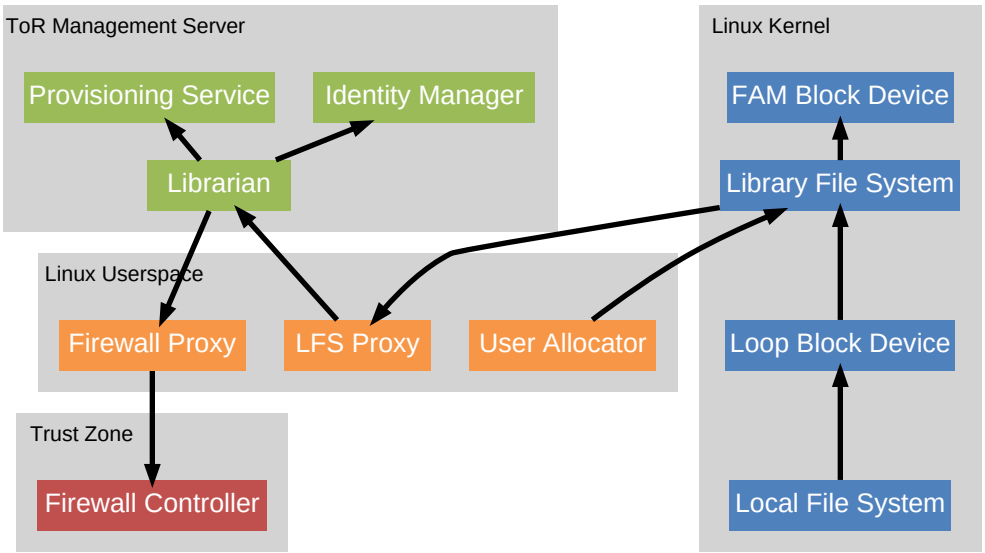
Managing storage within the machine poses some interesting challenges.

Each node runs a separate operating system instance. A single Load/Store Domain may have multiple authentication domains within it. This means that allocation and access control to the store will be managed separately from the node operating systems.

As hardware access control within the store is assigned on 8 GB boundaries, low-level storage allocation is done using the same boundaries. These allocation unit are called *Books*. A group of *Books* is called a *Shelf*. To provide smaller allocations, second level allocators may operate within the node operating system environment, parceling out bits of a shelf.

The overall storage software architecture within The Machine is summarized in the following diagram. Each piece of this diagram will be defined in the following sections.

Figure 9.1. Storage Architecture

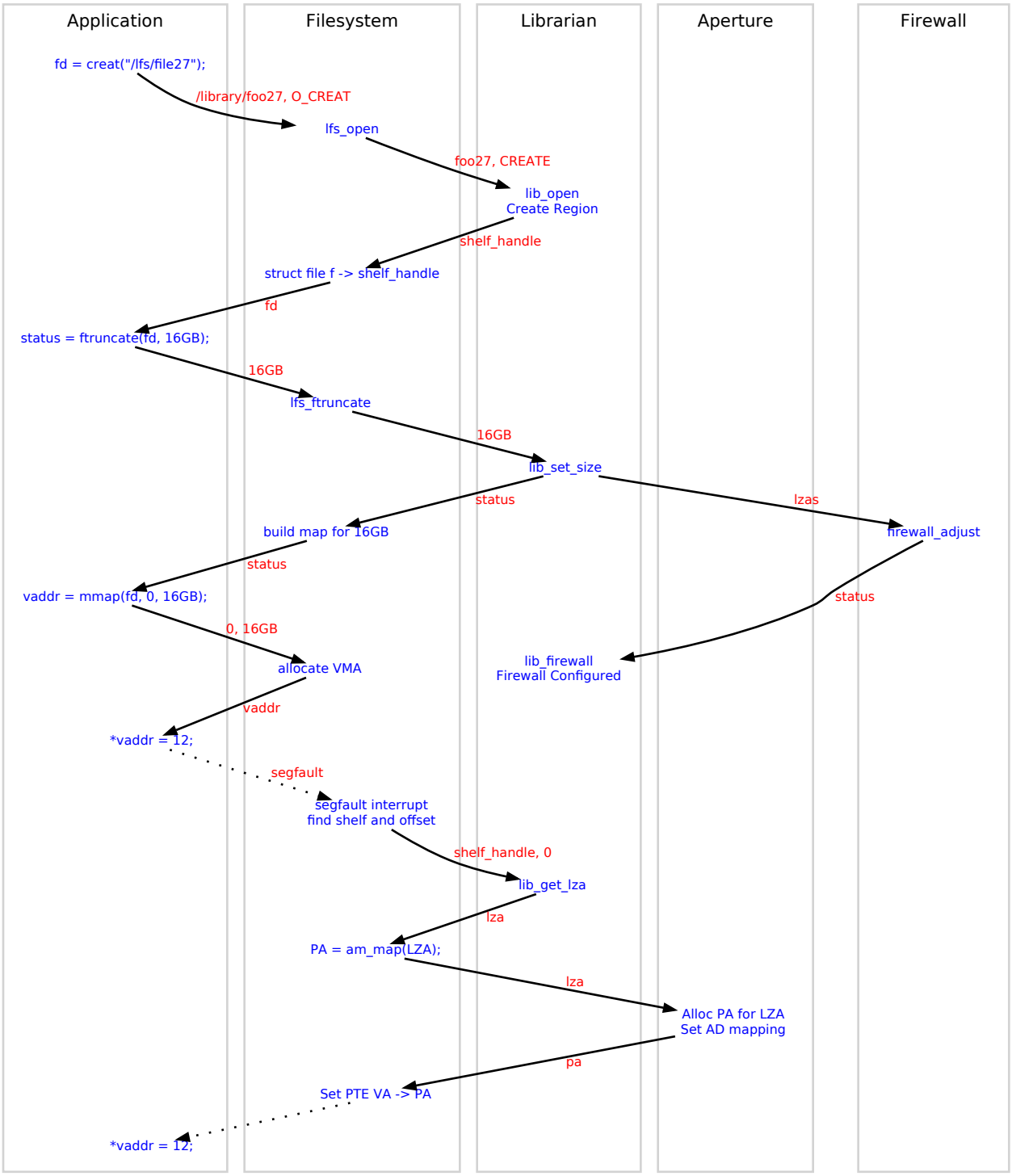


9.1. Storage Use Cases

To clarify how the storage architecture works together, a few simple examples are shown here.

9.1.1. Create new object, write a single byte

Figure 9.2. Create FAM Object



This example shows a single application creating a shelf in the library, allocating space, mapping and then finally storing a value.

The librarian checks permission and then creates a new shelf. It then returns a *shelf_handle* value back to the node. This handle will remain valid until the node closes the shelf, or the librarian discovers that the node has failed or rebooted.

That is true even if the region is deleted or renamed. A common POSIX idiom for allocating temporary anonymous storage is to create a file and immediately delete it, leaving the file descriptor as the only reference to it.

The next step is to allocate books within the shelf, which is done withe the ftruncate system call to set the size of the object.

For the node to access books within a shelf, the firewall within that node must be configured correctly. Logically, the entire management of all of the firewalls in the system is done within the librarian, but physically that hardware is located within the node. Hence the presence of a *Firewall Controller* that operates as a slave to the librarian within the node context.

There were two options discussed in the design review for how to manage the firewall:

1. Adjust firewall automatically as the set of books within a shelf changed.

2. Adjust firewall only in response to mapping requests.

We decided to have the firewall track the set of open books, and not the set of mapped books. We think this will result in fewer firewall modifications, hence within the diagram the firewall is adjusted when the shelf is resized. This decision may be revisited if we find that it causes too many unnecessary firewall modifications.

Not shown in the diagram is the fact that the librarian must wait for the firewall to be adjusted before returning the newly allocated book Logical Z addresses (LZAs) to the node.

Next, the application maps the entire shelf into its virtual address space. This causes no librarian interaction, only the allocation of a suitable VMA within the Linux kernel. There is no PA space allocated at this time.

Finally, the application stores a value through the mapping. Because there is no physical mapping, this causes a segmentation fault which eventually lands within the librarian file system code. That identifies the virtual address within the VMA, which references the related shelf_handle along with an offset within that shelf.

There was another design decision point here—when should we get the LZAs from the librarian for each book within the shelf. There are two options:

1. LZAs are automatically delivered whenever the set of books within the shelf changes.

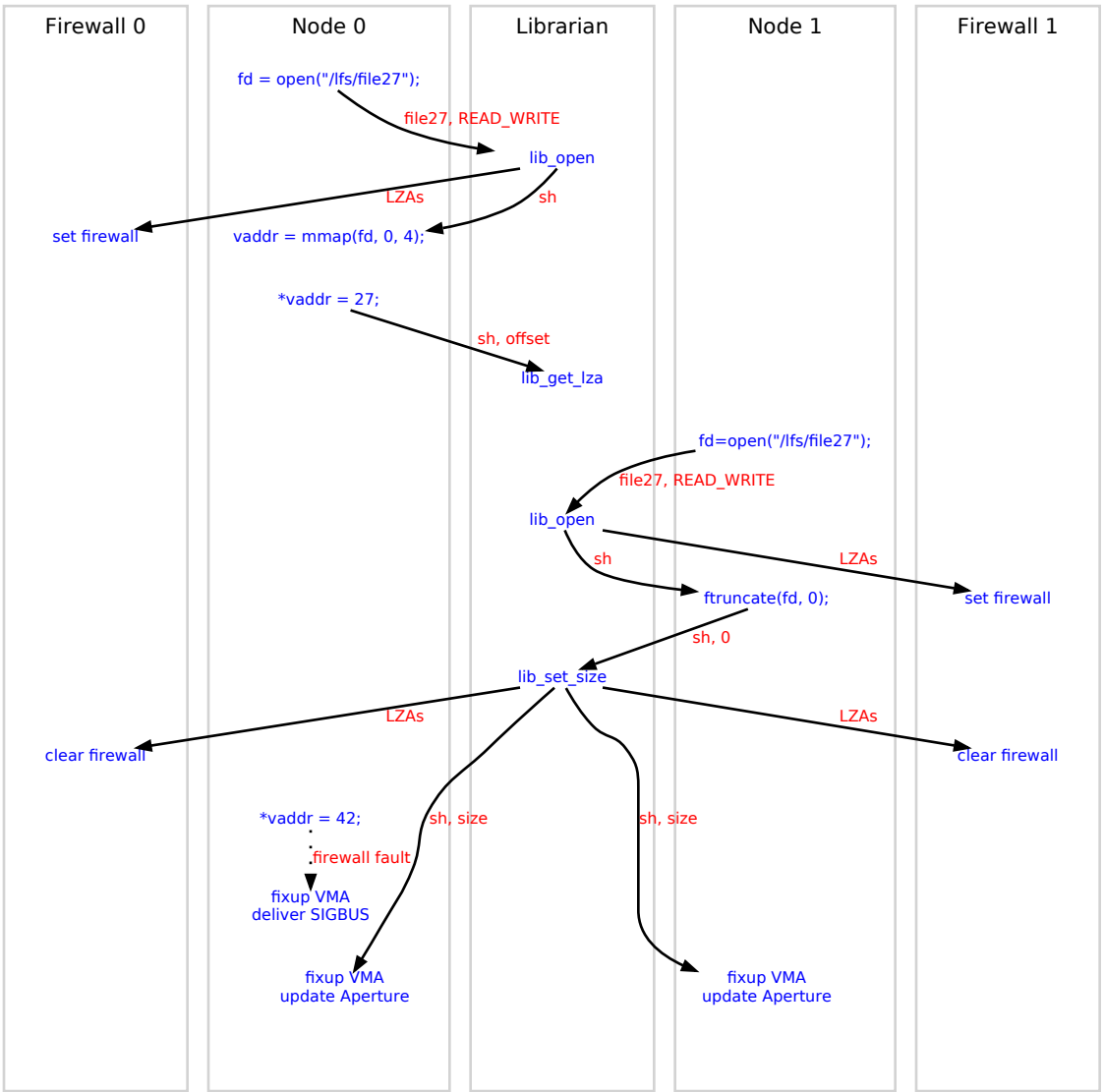
2. LZAs are fetched from the librarian as needed (and cached locally).

We decided to have the file system fetch the LZA values as needed. This decision may be revisited if latency of this operation is a problem.

So, with the shelf_handle and offset values known to the kernel, we pass those to the librarian which returns the LZA for the relevant book. That LZA is passed to the aperture/descriptor manager so that a physical address (PA) within the aperture can be allocated. That PA is then placed in the PTE for the faulting virtual address and the store instruction is restarted, this time it succeeds and the application proceeds from there.

9.1.2. External size change

Figure 9.3. External Region Size Change



9.2. Firewall Proxy (FP)

9.2.1. Requirements

Mediate between the firewall network interface and the firewall controller running in Trust Zone

9.2.2. Dependencies

- Kernel device for Firewall Control
- Trust-zone based Firewall Controller

9.2.3. Implementation

The Firewall Proxy will be written in Python and communicate with the Librarian using a TLS/JSON protocol. On the other side, it will communicate with the Firewall Controller, either within the Linux Kernel or running in Trust Zone.

9.3. Firewall Controller (FC)

9.3.1. Requirements

Configure firewall hardware under the direction of the Librarian

9.3.2. Dependencies

- Trust Zone to Linux communication

9.3.3. Hosting

During development, the FC will be hosted within the Linux system running on each node.

Once we have the ability to communicate between the trust zone and the librarian, the FC can move into the trust zone.

9.3.4. Discussion

The Firewall Controller provides the fundamental security infrastructure within The Machine, limiting access to LZAs within the LSD. It provides only the mechanism, and not the security policy. That policy lives solely within the Librarian.

Because the FC performs fundamental access control, the identity of the librarian must be ascertained, and all communications from that agent must be secured.

9.3.5. Initial Implementation Plan

To ease the development of the FC, it will initially be run within the node's Linux kernel environment. This will not offer the desired security characteristics, but will simplify development until we have a reasonably stable implementation of the FC and stable interfaces between the FC and the librarian.

9.3.6. Eventual Implementation Plan

At some point, the FC will transition into the trust zone to provide security within the store between nodes. This will require some new communication mechanism, either passing data through Linux and into the Trust Zone, or perhaps directly from the Trust Zone to the librarian.

9.4. Firewall Protocol (FP)

There are three separate architectural plans for the combination of Firewall Controller and Librarian, and each of these will involve a different interface between the two:

1. Librarian on the top-of-rack management server and FC in the Linux environment on each node.

   This can use a simple network link between the librarian and the FC, leveraging the same JSON toolchain as the LP.

2. Librarian on the top-of-rack management server and FC in the trust zone "underneath" the Linux environment.

   Assuming no direct network connection from the trust zone, this will involve a proxy within the Linux environment forwarding data from the librarian to the FC..

3. Librarian and FC both running in the trust zone.

   This is the easiest case, with the librarian and FC communicating via direct function calls.

For the first two cases, a network protocol will be necessary.

9.4.1. Requirements

Expose an interface from the Firewall Controller for the Librarian

Provide for secured and authenticated communication between the Librarian and the Firewall Controller through the Firewall Proxy.

9.4.2. Dependencies

- Firewall Controller

9.4.3. Hosting

While the librarian resides on the top-of-rack management server, the FP will need to offer a secure communications path between the librarian and the FC.

In the future, if the librarian ends up as a distributed application running within each node’s trust zone, then the communication between the librarian and the FC can be in the form of direct calls to a library.

9.4.4. Discussion

To secure the Firewall Protocol against attacks from within Linux, some authentication of the contents of each firewall command is required. Whether the contents need to be kept secret from Linux is yet to be determined.

9.5. Librarian

Each library shelf is be a collection of books allocated within the entire storage space of the LSD. Shelves can grow or shrink over time. Each shelf will have a nominal byte size.

Shelves will live in a naming hierarchy, operating like a traditional POSIX file system. Directories and shelf attributes are not necessarily stored within FAM.

Access to shelves will be controlled using the identity manager for authentication, with attributes stored within the librarian labeling authorization related to each principal.

9.5.1. Requirements

Manage global storage allocation across a single LSD in units of books (8 GB).

Manage firewall configuration for all participating nodes.

Allow applications to direct allocation to specific nodes within the system.

9.5.2. Resource Discovery

As described in the Provisioning chapter, many agents will use a static config file describing the topology and resources in the cluster.

The Librarian team has chosen the classic "INI" file definition. [Section] headers are delineated in square brackets and section content is expressed as "key=value" pairs. This format provides sufficient flexibility to fully describe the topology of The Machine. A support program in the librarian suite reads an INI file and constructs the initial SQL database for the Librarian.

The Full Rack Demo (FRD) definition specifies the following maximal configuration: * Single rack (numbered "1") * Up to eight enclosures per rack (1 through 8) * Up to ten nodes per enclosure (1 through 10)

In a simplified coordinate system, a node can be uniquely identified by the tuple (rack, enclosure, node) or a shorthand "R:E:N". Furthermore, in the FRD, there can be a maximum of 80 nodes. Simple math can map an R:E:N location to a value 1 - 80, i.e., 1:3:5 is node 25.

Topology details for The Machine include:

- Nodes
  - Location coordinates (rack:enclosure:node)
  - MAC address
  - Total NVM on the board (i.e., behind each Psylocke)
- Interleave Groups
  - Number
  - Psylocke membership (R:E:N:CID of each Psylocke)

Simplified "Extrapolation" INI file

The FRD definition hardcodes Interleave Groups to be all four Psylockes on any given node. Thus in an eighty-node Machine there will be eighty Interleave Groups. This suggests the simplified INI format below:

```
# For a small, emulated machine.
[global]
node_count = 4
book_size_bytes = 8M
nvm_size_per_node = 512B
```

The global section is always needed. This short form is based on the 1:1 assumption of node:interleave group mentioned earlier. The three values shown provide enough information for the construction program to extrapolate all other data; explicit mention of interleave groups is not required.

Note the books size of eight megabytes. The L4TM team is using fabric attached memory emulation as a design platform, typically running on a laptop. The book size is appropriate for the backing store that would be present on such a platform.

The NVM per node is given in terms of books. Thus the INI file above describes four nodes of 512 books per node, similar to a real machine node. With a book size of 8M, emulated NVM per node is 4G, for a total of 16G for this "Machine".

A more generic INI file with arbitrary interleave group definitions will be documented soon.

9.5.3. Book allocation policies

While the FAM store may be treated as a single, uniform entity, there are performance and reliability goals which may be best served by providing some control in where books are allocated for each shelf.

The Machine hardware addresses books as offsets in an *interleave group*, not offsets on a node. The media controllers can be configured, in general, to direct the hardware to interleave NVM access across nodes. The default 1:1 interleave group:node definition is a corner case but makes for the simplest conceptual use. An interleave group "equals" a node.

The Librarian File System (LFS) seen on each node allows software to change allocation policy by working with POSIX "extended attributes". These are generic key-value pairs of metadata that can be assigned to any file (shelf). See the man pages for setfattr(1) and getfattr(1) from the "attr" package on Debian/L4TM.

The Librarian will allow applications to direct allocations by setting the user.interleave_request extended attribute. This is a numeric attribute with one byte for each book to specify the interleave group desired for that book in the shelf. For example, if you want to create a four-book file called *foo* that alternates between interleave group 2 and 3, you'd do:

```
$ touch /lfs/foo
$ setfattr -n user.interleave_request -v 0x02030203 /lfs/foo
$ fallocate -l 32G /lfs/foo
```

For regions of the shelf not described in this attribute (and when user.interleave_request is not set at all), the Librarian relies on another extended attribute to guide allocations. A look at the other intrinsic attributes is in order: create a shelf, then dump all the extended attributes associated with that shelf:

```
$ touch /lfs/bar
$ getfattr -d /lfs/bar
getfattr: Removing leading '/' from absolute path names
# file: lfs/bar
user.LFS.AllocationPolicy="LocalNode"
user.LFS.AllocationPolicyList="LocalNode,RandomBook,Nearest,LZAascending,LZAdescending"
user.LFS.Interleave
```

The default policy, "LocalNode", is based on the 1:1 node:interleave group configuration. Only books from the node of the requesting process will be allocated. Obviously this policy loses meaning if arbitrary interleave group definitions are used.

Available allocation policies are enumerated in "user.LFS.AllocationPolicyList":

- LocalNode: only books from the local node/IG are used. When all the books in the interleave group equated to that node all allocated, further allocations fail.
- RandomBook: every book is chosen at random across the cluster without regard to location. Implicit weighting ensues, statistically favoring interleave groups with more available books.
- Nearest: start with LocalNode, then allocate randomly from nodes in the same enclosure, and finally try RandomBook.
- LZAascending: LZA means "Logical Z Address", a property of books in an interleave group. This policy starts at the lowest-numbered interleave group and works up. Essentially, books from node one, then node two, etc.
- LZAdescending: start at the highest-numbered interleave group and work down.

Figuring out which interleave groups to use is left as an exercise for the application. Note that some policies (LZAascending, LZAdescending) lead to highly imbalanced book selection and may not represent a sane choice for performance considerations.

Applications can query the actual location of each book by reading the user.LFS.Interleave extended attribute. That will contain one byte for each allocated book in the shelf naming the interleave group that book is contained in.

For example, if our emulated machine is using eight megabyte books, a size of 20M needs three books:

```
$ truncate -s 20M /lfs/bar
$ getfattr -e hex -n user.LFS.Interleave /lfs/bar
getfattr: Removing leading '/' from absolute path names
# file: lfs/bar
user.LFS.Interleave=0x030303
```

The three books came from interleave group 3. Trivia question: this machine emulation was configured with the 4-node extrapolation INI file discussed above, and the current policy was "LocalNode". On which node were the above commands executed?

9.5.4. Permissions and Access Control

Each node will be assigned to a particular tenant, and each tenant has an identity. Operations within the library file system will be governed solely by the tenant of the node running the application.

Each shelf will be *owned* by a tenant; permissions can be set to allow other tenants to access the shelf by setting the *other* access mode bits.

At least for now, there are no *groups*, so the group access control bits will not be useful.

The Librarian may eventually want to include finer-grained access control. Within the Linux environment, that would use the user-id and group-ids for the process. Those node-specific values would then be translated into global identities by the Library File System code on the node, using the Identity Manager to convert the local values into global identities and passing those to the Librarian.

9.5.5. Dependencies

- Identity Manager
- Firewall Controller
- Manageability Software

9.5.6. Hosting

For current hardware platforms, the top-of-rack management server

For future hardware platforms, a distributed system running within a secure enclave or trust zone.

9.5.7. API

The librarian interface will be designed to provide POSIX semantics along with extensions necessary to support direct mapping for I/O to the store. This may be derived from the prototype librarian interfaces.

The initial librarian will offer a JSON interface over the local network using the Librarian Protocol (LP) which will be secured using TLS1.2.

The librarian will require an interface to the Firewall Controller running on the individual nodes.

The librarian will require an interface to the manageability software to provide notification about hardware and software status (failed memory modules, perhaps some level of accounting information?)

The librarian will also offer a RESTful API to query the state of the librarian by monitoring and logging tools elsewhere in the system.

9.5.8. Initial Architectural Plans

The librarian will initially be implemented as a single, centralized application running on the top-of-rack management server. Communication with the operating system instances on each node will occur over the local network.

The meta data for the store will be maintained solely within the top-of-rack server environment. It will be robust in the face of hardware and/or software failures. To meet performance targets, this will likely require SSD storage within the top-of-rack management server. The current plan is to store data in a SQL Lite database.

Performance will be in the range of 1000 meta-data operations per second. Each of those operations requires a network transaction between the Node OS and the Librarian along with database transaction within the Librarian itself.

9.5.9. Aspirations

Our future plans have the librarian operating as either a centralized replicated or distributed service, running within the machine environment on the nodes. To offer security, the librarian would run in a protected environment on the node, either within the trust zone, other secure enclave or using some of the nodes as management infrastructure.

Communication between librarian agents would occur both directly through the store, using Atlas-like data structures, and potentially over the local network.

The store would be self-defining, meaning that no external data would be required to describe the contents of the store itself. Metadata operations within the store would be robust in the face of hardware failures. Robustness of data operations would be under the purview of the higher software levels.

9.6. Library File System Proxy (LFSP)

To avoid using TLS and JSON in the kernel and simplify the development process, the Library File System and Librarian communicate through the Library File System Proxy (LFSP), a Linux user space application written in Python. The kernel interface for this is a modified version of the FuSE protocol. The Librarian interface uses JSON over TLS.

9.6.1. Requirements

Bridge the binary FuSE protocol and TLS/JSON Librarian protocols

9.6.2. Dependencies

- Library File System
- Librarian Protocol
- Librarian

9.6.3. Hosting

The LFSP runs in Linux user space on each node. It is written in Python.

9.7. Librarian Protocol (LP)

The Librarian Protocol (LP) defines the interface between the global Librarian and the node operating system.

9.7.1. Requirements

Provide secure and reliable communications between the Node OS and the Librarian.

Implement POSIX file system semantics, with extensions as necessary to manage direct FAM mappings.

9.7.2. Dependencies

- Identity Manager

9.7.3. Hosting

For current hardware platforms, a link between the top-of-rack management server and the operating system running on each node.

For future hardware platforms, an API between the node OS and the distributed librarian running within the secure computing environment on the same node.

9.7.4. API

This will mirror the librarian API on the other end of a network connection. There will be additional API elements as needed to manage the communications link, including identity and authentication management.

9.7.5. Initial Architectural Plans

The Librarian Protocol will use JSON, mirroring the contents of the FuSE API from the kernel with additions as needed to support FAM.

9.7.6. Aspirations

When the librarian becomes a distributed service running within the secure environment on each node, the LP may be used to communicate between the node

OS and this trusted environment, if that communication needs to be reduced to a duplex byte stream.

9.8. Librarian Monitoring Protocol (LMP)

The Librarian Monitoring Protocol provides information to monitoring services about the state of the overall storage system. It is purely a query service; no state changes are possible over this interface.

The Librarian Monitoring Protocol is provided within the Librarian as an HTTP server that accepts HTTPS connections to receive requests and perform operations via a REST interface. This outward interface has the following qualities:

- HTTPS 1.1 (TLS)
- Representation State Transfer (REST)
- JavaScript Object Notation (JSON) is used to exchange models and state
- UTF-8 encoding
- The current version of this protocol is 1.0

To allow for changes in the protocol, the client should transmit the version it is using in the request's Accept header:

- Accept: application/json; version=1.0

Based on the above qualities, every PUT and GET request should have the following HTTP header, which describes the format of the data contained in the body of the request:

- Content-Type: application/json; charset=utf-8; version=1.0

The interface supports only a single HTTP methods: GET. This document describes the external API using these methods.

Although the methods described in this document show the possibility of the HTTP response code "401 Unauthorized", the Librarian does not yet support authorization; the intent is to add support for authorization in a future release.

9.8.1. Global Information

This request provides information about the state of the librarian without any node-specific details.

Global Memory Information

Description

Get global memory usage information

Resource

/global

HTTP Method

GET

Response

200 OK: A JSON structure containing global information of the form:

```
{
        "memory": {
                "total": 351843720888320,
                "allocated": 109951162777600,
                "available": 219902325555200,
                "notready": 21990232555520,
                "offline": 0
        },
        "socs": {
                "total": 80,
                "active": 62,
                "offline": 18
        },
        "pools": {
                "total": 320,
                "active": 300,
                "offline": 20
        },
        "active": {
                "shelves": 4000,
                "books": 32000,
        }
}
```

Table 9.1. Global Librarian Information

Name	Contents
memory.total	Total memory visible to librarian
memory.allocated	Memory in allocated shelves
memory.available	Memory available for allocation
memory.notready	Memory which is being prepared for allocation
memory.offline	Memory which is unavailable for allocation
socs.total	Total socs known to librarian

Name	Contents
socs.active	Socs currently registered to librarian
socs.offline	Socs known but not registered
pools.total	Total memory pools known to librarian
pools.active	Memory pools available for use
pools.offline	Memory pools not available for any reason
active.shelves	Shelves open by any SoC
active.books	Books associated with an open shelf.

In this context, an “SoC” is a compute element within The Machine comprising local DRAM, an SoC and the NGMI switch connecting it to the fabric. A “pool” is the smallest unit of memory which can independently report failures; in the current hardware this is a Psylocke and associated memory.

“notready” memory has not yet been erased for use.

“offline” memory is memory which has been taken out of service due to a failure.

List of Nodes

This provides the librarian’s view of the current node configuration. This is a mirror of the global node configuration information and is provided through this API as a convenience.

Description

List nodes in The Machine instance managed by the Librarian.

Resource

/nodes

HTTP Method

GET

Response

200 OK: A JSON structure containing node information from the configuration database, adapting the *node* element as described in Nodes:

```
{
        "nodes":
        [
                {
                        "coordinate": "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/er
```

```
                        "serialNumber": "<serial number for node>"
                        "soc":
                        {
                                …
                        },
                        "mediaControllers":
                        [
                                …
                        ]
                }
        ]
}
```

The soc element is as described in SoC Configuration Data. The mediaControllers section is as described in Psylocke Configuration Data.

Memory Configuration

This provides the librarian's view of the current memory configuration. This is a mirror of the global interleave group configuration information and is provided through this API as a convenience.

Description

List interleave groups in The Machine instance managed by the Librarian.

Resource

/interleaveGroups

HTTP Method

GET

Response

200 OK:

A JSON structure containing interleave group information from the configuration database, comprising the *interleaveGroups* element as described in Section 5.4.2, "Memory Configuration":

```
{
        "interleaveGroups":
        [
                {
                        "groupId": 1,
                        "baseAddress": 0,
                        "size": 4398046511104,
                        "mediaControllers":
                        [
                                "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                        enclosure/1/node/1/memory_board/1/media_controller/1",
```

```
                                "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                        enclosure/1/node/1/memory_board/1/media_controller/2",
                                "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                        enclosure/1/node/1/memory_board/1/media_controller/3",
                                "machine_rev/1/datacenter/pa1/frame/A1.above_floor/rack/1/
                                        enclosure/1/node/1/memory_board/1/media_controller/4"
                        ]
                },
                …
        ]
}
```

9.8.2. Memory Utilization Information

The Librarian can report both memory allocations and active memory usage information. Active memory usage measures the amount of memory currently open by an SoC. Allocations are reported per Psylocke, current usage per Psylocke and per SoC.

Memory Allocations

Description

This reports allocations across The Machine.

Resource

/allocated/{coordinate}

HTTP Method

GET

Input

coordinate: subset of the machine to report allocations about. It can be the coordinate of a rack, enclosure, node, memory board or media controller. In each case, the data reported will reflect allocations within that portion of the machine. It may also be the coordinate of the full machine, in which case the values reported will be the same as would be reported in the /global API for the same fields.

Response

200 OK: A JSON structure containing memory allocation information within the target portion of The Machine of the form:

```
{
        "memory": {
                "total": 137438953472,
                "allocated": 68719476736,
```

```
                "available": 34359738368,
                "notready": 17179869184,
                "offline": 17179869184
        }
}
```

Table 9.2. Memory Allocation Information

Name	Contents
memory.total	Total memory visible to librarian
memory.allocated	Memory in allocated shelves
memory.available	Memory available for allocation
memory.notready	Memory which is being prepared for allocation
memory.offline	Memory which is unavailable for allocation

“notready” memory has not yet been erased for use.

“offline” memory is memory which has been taken out of service due to a failure.

Memory Activity

Description

This reports memory access by SoCs across The Machine.

Resource

/active/{coordinate}

HTTP Method

GET

Input

coordinate: SoC to report information about, or the coordinate of the machine itself, in which case the values reported will be the same as would be reported for the same elements of the /global response.

Response

200 OK: A JSON structure containing memory activity within the target portion of The Machine of the form:

```
{
        "active": {
                "shelves": 4000,
                "books": 32000,
```

```
        }
}
```

Table 9.3. Memory Activity Information

Name	Contents
active.shelves	Shelves open by any SoC
active.books	Books associated with an open shelf.

9.8.3. Shelf Information

The librarian can report allocations and activity on each shelf in the file system.

Directory Information

Description

Get directory information

Resource

/shelf/{pathname}

HTTP Method

GET

Input

pathname: Full path of the directory within the librarian file system (not including the /lfs prefix used within the nodes). For the FRD, there are no sub-directories, so this must be empty.

Response

200 OK:

```
\       {
\               "owner": 237,
\               "group": 42,
\               "mode": 0775,
\               "entries":
\               [
\                       {
\                               "name": "file1",
\                               "type": "file",
\                               "owner": 237,
\                               "group": 42,
\                               "mode": 0644,
```

```
\                               "size": 23203984720,
\                       },
\                       {
\                               "name": "file2",
\                               "type": "file",
\                               "owner": 237,
\                               "group": 42,
\                               "mode": 0775,
\                               "size": 23203984720,
\                       },
\                       …
\               ]
\       }
```

Shelf Information

Description

Get shelf information

Resource

/shelf/{pathname}

HTTP Method

GET

Input

pathname: Full path of the shelf within the librarian file system (not including the /lfs prefix used within the nodes)

Response

200 OK:

```
{
        "owner": 237,
        "group": 42,
        "mode": 0644,
        "size": 23203984720,
        "booksize": 8589934592,
        "active":
        [
                "SoC 0 coordinate",
                "SoC 27 coordinate",
        ]
        "books":
        [
                98432750932,
                203987503298475325,
                …
        ]
}
```

books contains the list of LZAs for the books in the shelf.

Book Information

Description

Get information about all books

Resource

/books/{interleave-group}

HTTP Method

GET

Input

interleave-group: (optional) Interleave group to limit book list for. If not present, information for all interleave groups is returned.

Response

200 OK:

```
{
        "book-size": 8589934592,
        "books" :
        [
                {
                        "lza": 429496729600,
                        "state": "allocated",
                        "shelf": "file1",
                        "offset": 98432750932,
                },
                {
                        "lza": 9878424780800,
                        "state": "available",
                },
                …
        ]
}
```

Table 9.4. Book States

Name	Contents
allocated	Book is assigned to a shelf
available	Book is available for allocation
notready	Book is being prepared for allocation
offline	Book is unavailable for allocation

Allocated books will have the shelf and offset fields set. Available, notready and offline books will not.

While this response doesn’t explicitly list the interleave group, that value can be computed from the interleave group information and the lza of each book.

9.9. Aperture Manager (AM)

FAM is referenced globally by Logical Z address (LZA). LZAs are mapped into the processor physical addresses (PAs) by the Address Mapping Unit within the GenZ hardware. The Aperture Manager manages the mapping between LZAs and PAs.

Unlike traditional CPU page tables, GenZ separates the notion of access control (Firewall) from address mapping (Address Mapping Unit). Access control is done outside of the node operating system in the Firewall Controller, but for simplicity and performance reasons, the mapping portion is done inside the node operating system.

The address mapping unit operates on two different object sizes space — full books (8 GB) and small booklets (64kB). The AM interface will support mapping of both sizes.

Unmapping LZA ranges must be preceded by flushing and invalidating the processor cache in the matching PA range.

9.9.1. Requirements

Manage PA range devoted to FAM access

Handle required interactions with the Linux VM subsystem.

Handle cache flushing as required when modifying Z to P mappings.

Offer interfaces for PA range cache flushing and invalidating for other modules within the kernel.

9.9.2. Dependencies

- None

9.9.3. Discussion

For environments where the LZA space is larger than the PA space, applications which use the full LZA space range will require careful management of the mapping to avoid performance problems.

9.9.4. Hosting

AM will be implemented as a Linux kernel module.

9.9.5. API

AM will operate on LZA ranges, not shelf names. Calling modules will be required to compute LZAs themselves.

9.10. Library File System (LFS)

The Library File System exposes the whole librarian store as a standard Linux file system, offering all of the existing file system tools for managing naming and access control.

Identity within the LFS will be controlled by the Identity Manager. As the current plan for that is to identify only tenant/other, LFS files owned by the node tenant will appear to be owned by *root* (UID 0) while files owned by other tenants will appear to be owned by *nobody* (UID 65534).

In the default configuration, LFS will be mounted on /lfs.

9.10.1. Requirements

Expose the Librarian storage structure to Linux applications as a file system

1. Offer direct-mapping to librarian objects using DAX
2. Provide a mechanism to deliver LZA ranges to the Library Block Device (LBD)
3. Support atomic operations via ioctls on LFS files.

9.10.2. Dependencies

- Librarian
- Identity Manager
- Aperture Manager

9.10.3. Hosting

The Library File System will be implemented as a Linux file system within the kernel. It will provide DAX so that applications can directly map FAM by using mmap on a LFS file object.

9.10.4. API

The Linux VFS layer defines most of the exported API necessary. In addition to that, the LFS will provide interfaces for the Library Block Device as needed.

Applications will use the standard POSIX APIs to open and map LFS file objects.

Atomics will be exposed to applications via the fam_atomic library. That library uses ioctls (defined in the fam_atomic header) to talk to the LFS kernel module which will then read and write the atomic registers within the Z-bridge.

9.10.5. Initial Architectural Plans

For prototyping purposes, the LFS will be implemented as a FuSE file system. That should offer the ability to test much of the LFS, librarian and LP design without needing to operate in kernel mode.

The user-mode pieces talking LP will be written by Python, leveraging existing FuSE and JSON python libraries.

We'll then fork the FuSE code in the kernel to allow us to perform the necessary operations to manipulate FAM and perform atomics, and slowly mutate the system to provide all of the necessary operations.

9.10.6. Simplifying Assumptions

We don't expect that most applications running on The Machine will require that LFS be a complete POSIX-compliant file system. Here are a few details about what we're implementing for the current version:

ctime, mtime, atime

: These values will always be left at 00:00 UTC 1-Jan-1970

uid, gid

: These are stored in the Librarian as uninterpreted numeric values. All semantics relating to users and groups will be evaluated within the Linux environment on the SoC. Uid and gid allocation on each SoC must be coordinated to ensure they match where necessary.

sparse files

: In most POSIX file systems, if you writing to only widely separated locations within the file, areas within the file which are not ever written will not have any storage allocated for them. For LFS, we're not supporting this, preferring to allocate storage when the file size is set and not when data are eventually written.

software mirroring

In supporting direct access to FAM, we have eliminated the ability to provide data redundancy at the file system level. If redundancy is required, applications must implement that in user space.

direct mapping

The whole point of TM is for applications to have load/store access to FAM. LFS provides this by mapping FAM through the aperture and then through the SoC page table system directly into the application's address space.

sub directories

LFS won't support arbitrary sub directories, so plan on creating all files in a single directory. We're still sorting out whether that will be /lfs itself, or potentially /lfs/<tenant> as a way to perform tenant-based access control.

special files

Devices, UNIX domain sockets and named pipes are not supported in LFS.

read/write

Read and write operations to FAM files will work as expected. Data read/written to FAM files will be visible immediately through FAM mappings. fsync(2) can be used to ensure that the contents are flushed all the way to FAM.

symlinks

LFS does not support symlinks.

hard links

LFS does not support hard links.

9.11. Library Block Device (LBD)

Existing local file systems implementations in the kernel, like Ext4, XFS and FAT, communicate with underlying storage through the Linux Block I/O Layer. This interface provides a small interface offering the ability to access storage in fixed size units (oddly, called *blocks*).

Traditional storage is accessed by copying data between the media and memory. A huge amount of the existing kernel infrastructure is devoted to managing these operations, attempting to make the movement of data on and off storage devices efficient.

In version 4.0, the Linux kernel added a set of interfaces to the block layer called Direct Access (DAX) . For storage which exists within the nodes PA space (such as FAM), DAX offers the ability to skip all of the data motion and reference storage directly.

9.11.1. Requirements

Provide a standard Linux block device interface on top of the Librarian File system.

An as-yet-unspecified configuration mechanism will exist that tells the kernel to create a block device from a specific shelf.

Support DAX interfaces for file systems built on top of a LBD.

9.11.2. Dependencies

- Library File System
- Librarian
- Aperture Manager

9.11.3. Hosting

The Library Block Device will be a standard Linux kernel driver.

9.11.4. Initial Architectural Plans

Functionally, the LBD is "the same" as the existing kernel loop block device driver. That driver does not have DAX support, and the current DAX block device interface cannot be supported on top of LFS. So, the first step is to start modifying the DAX block driver interface to allow this to happen.

For now, we can simply use the existing loop block device driver. That does not provide direct mapping support, making this significantly slower than LFS.

This also means that FAM atomics will not work in file systems mounted on top of LBD.

9.12. Retail Memory Broker (RMB)

Because the unit of allocation offered by the librarian is large (8 GB), applications directly using that storage will presumably want some way to break shelves up into smaller pieces.

There may be more than one retail memory broker, split into two classes:

1. Single-node RMB

   Single-node RMBs will not deal with concurrency and caching issues outside of the scope of the node operating system. Local Linux file systems are effectively retail memory brokers.

2. Multi-node RMB

   Multi-node RMBs will handle simultaneous access to the shelf from more than one node. This will involve using the in-FAM locking facilities of the Atomics API.

9.12.1. Dependencies

- Library File System
- Atomics API.

9.12.2. Hosting

Retail Memory Brokers will be implemented in Linux user space on top of the Library File System.

9.13. Local File System (FS)

While it might be nice to think of the machine as a single unified computing environment, our existing Linux ecosystem assumes some per-OS-instance storage, so we want to allow a file system which isn't shared with any other node operating system.

The real reason for using a purely local file system is that we don't have a concurrent file system, nor will we have one any time soon. So, for purely practical reasons, we need to provide support for existing file systems so that we can install a Linux distribution and run applications.

Traditional Linux file systems are built on top of the Linux block layer, so we'll be offering that interface and planning on just using existing Linux file systems for local file systems.

9.13.1. Dependencies

- Library Block Device

9.13.2. Hosting

Any existing Linux file system should work on top of the LBD.

9.13.3. Initial Implementation Plan

Use the loop block device driver to provide a way to layer any arbitrary file system on top of a shelf.

Note that the loop driver does not currently support DAX, and so, at least for now, files in a local file system will not be directly mapped from the underlying FAM. Future work to provide a block driver which can support DAX on top of LFS will be required.

9.14. Concurrent File System (CFS)

The machine provides a unique environment for file system work. It looks a lot like a distributed computing system, and yet, because of the shared nature of the FAM, it also looks a lot like a concurrent multi-processor system. In reality, the machine offers a mixture of the two environments:

- Shared FAM can lead to shared in-FAM data structures for concurrent development models
- Separate operating system and security domains mean providing some level of global naming and access control, outside the purview of any one node operating system.
- Separate process spaces require additional synchronization and communication primitives to manage data structures with multiple accessors.

While the LFS offers a shared file system within FAM, it targets large-scale allocations, and is tightly tied in to the firewall mechanisms providing hardware security between the nodes.

Many applications within the machine environment will want to take advantage of smaller allocation units and higher performance, while still offering shared file access across nodes.

For this purpose, a concurrent retail file system is needed. At this point, we have no potential implementations selected, leaving us with a significant opportunity for research and development.

9.14.1. Dependencies

- Library File System

9.14.2. Hosting

A kernel-level file system built in to Linux for the machine.

9.15. Examples of Using FAM from Linux Applications

The librarian manages all of FAM for the entire set of nodes in the LSD. Each node accesses the librarian through the library file system, which is conventionally mounted at /lfs. Applications may create files within the library file system; all of the data for those files will be stored within FAM and all of the file names will be visible to other nodes in the LSD. By creating a standard Linux file system, no special API or library is required to access FAM.

FAM is accessed from Linux applications through calls like open, ftruncate and mmap. This section shows two short examples of accessing FAM from within a Linux application.

9.15.1. Example 1. Create a new FAM object, write a single byte.

```
/* Example 1: writing a byte to a new library file */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define GB      (1024l * 1024l * 1024l)

int
main (int argc, char **argv)
{
        int     fd;
        int     status;
        char    *vaddr;

        /* Create a new file in the library file system */
        fd = open("/lfs/file27", O_CREAT|O_RDWR, 0666);

        /* Set the size to 16GB */
        status = ftruncate(fd, 16 * GB);
        if (status)
                perror("ftruncate");

        /* map the entire file into our address space */
        vaddr = mmap(NULL, 16 * GB, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        if (vaddr == MAP_FAILED)
                perror("mmap");

        /* Write 12 to the first byte of the file */
        *vaddr = 12;

        return 0;
```

```
}
```

9.15.2. Example 2. Truncate a file and deliver SIGBUS

```
/* Example 2: resize a file from another node */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define GB      (1024l * 1024l * 1024l)

int
main (int argc, char **argv)
{
        int     fd;
        int     status;
        char    *vaddr;

        /* Open an existing file in the library file system */
        fd  = open("/lfs/file27", O_RDWR, 0666);

        if (!argv[1]) {
                /* No argument for node 0 */

                /* map the entire file into our address space */
                vaddr = mmap(NULL, 4, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
                if (vaddr == MAP_FAILED)
                        perror("mmap");

                /* Write 27 to the first byte of the file */
                *vaddr = 27;

                /* now pause, waiting for node 1 to run */
                printf ("Waiting...");
                (void) getchar();

                /* Write 42 to the first byte of the file */
                *vaddr = 42;
        } else {
                /* Any argument for node 1 */

                /* Set the size to 0 */
                status = ftruncate(fd, 0);
                if (status)
                        perror("ftruncate");
        }

        return 0;
}
```

Chapter 10. Atomics API

Provides access to the fabric attached memory atomic mechanisms.

The atomics mechanism within the fabric provides for atomic operations to fabric attached memory. Hardware is provided to allow each core within the SoC to simultaneously execute an atomic operation via a set of registers within the zbridge SoC interface.

These will be exposed to user space for files within the librarian or concurrent filesystems as a set of ioctl operations.

10.1. Cache-line alignment

Because the atomics hardware works directly on FAM, without regard to the state of the SoC caches, any SoC load/store operations within the same cache line will interact with atomic operations, even where those operations are not to the same memory locations.

To make atomic usage easier, two sets of APIs are included: one set uses the atomic datatypes directly, and the second set wraps those datatypes in structs that are padded and intended to be aligned to a cacheline boundary.

10.2. Locking API

In addition to the primitive hardware operations, the library includes locking operations based on the atomic operations.

10.3. Initial Implementation

The initial implementation will specify a set of ioctl operations within the librarian filesystem that perform the full set of operations specified in the chipset ERS Global synchronization chapter.

The kernel will not perform the cache flushing or invalidation necessary to make these operations consistent with direct load/store operations. The application must perform them if it chooses to directly access atomic values.

Chapter 11. Cache Flushing and Invalidation API

In a normal file mapping environment, the contents of memory are not guaranteed to be persistent until the application calls msync. A direct-access memory mapping to non-volatile storage allows the application to directly flush cached memory data and avoid the expense of a system call.

Separately, shared access to memory from processes not in the same cache coherence domain requires that cache flushing and cache invalidation operations be added to make memory changes visible to the application.

Intel has created a library which addresses the first problem, called libpmem; the web page for that is http://pmem.io/nvml/libpmem/. We will be using that library for performing cache flushing and will extend it to perform cache invalidation as well.

11.1. Providing libpmem on Arm64 for The Machine

libpmem is currently an x86-only library. This need to be ported to Arm64.

libpmem also includes various memory copy and clear functions that incorporate cache flushing and draining semantics. We will port those functions to operate correctly, but do not currently plan on optimizing them using non-temporal instructions like the current x86 implementations.

11.2. pmem.io libraries

Here’s a complete list of the pmem.io libraries and a short description. This includes libpmem itself, the low-level management API along with additional libraries built on top of that.

We are not planning on supporting libraries other than the basic libpmem within L4TM. However, as they purportedly use libpmem for all hardware abstraction, it’s quite possible that they could be supported if necessary at some point.

libpmemobj

> The libpmemobj library provides a transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming. Developers new to persistent memory probably want to start with this library.

See the libpmemobj [http://pmem.io/nvml/libpmemobj] page for documentation and examples.

libpmemblk

The libpmemblk library supports arrays of pmem-resident blocks, all the same size, that are atomically updated. For example, a program keeping a cache of fixed-size objects in pmem might find this library useful.

See the libpmemblk [http://pmem.io/nvml/libpmemblk] page for documentation and examples.

libpmemlog

The libpmemlog library provides a pmem-resident log file. This is useful for programs like databases that append frequently to a log file.

See the libpmemlog [http://pmem.io/nvml/libpmemlog] page for documentation and examples.

libpmem

The libpmem library provides low level persistent memory support. The libraries above are implemented using libpmem. Developers wishing to roll their own persistent memory algorithms will find this library useful, but most developers will likely use libpmemobj above and let that library call libpmem for them.

See the libpmem [http://pmem.io/nvml/libpmem] page for documentation and examples.

libvmem

The libvmem library turns a pool of persistent memory into a volatile memory pool, similar to the system heap but kept separate and with its own malloc-style API.

See the libvmem [http://pmem.io/nvml/libvmem] page for documentation and examples.

libvmmalloc

The libvmmalloc library transparently converts all the dynamic memory allocations into persistent memory allocations. This allows the use of persistent memory as volatile memory without modifying the target application.

See the libvmmalloc [http://pmem.io/nvml/libvmmalloc] page for documentation and examples

Chapter 12. Application Programming Interfaces

This section documents the public APIs that are either specific to using The Machine, or existing POSIX/Linux APIs which take on special importance in The Machine.

12.1. File system calls, the Library File System (LFS) and Fabric Attached Memory (FAM)

The Library File System (LFS), which is built directly on top of Fabric Attached Memory (FAM), offers enough of a POSIX-like environment for our applications to work correctly, but there are some subtle limitations and other differences which have simplified the design of the overall system.

The Library File System is conventionally mounted at /lfs.

This section documents POSIX and other Linux APIs which are expected to be used with LFS, describing their effect as related to LFS and documenting any limitations from standard Linux.

12.1.1. ftruncate(int fd, off_t length)

This is the easiest way to set the size of an LFS file. For LFS, we don’t support sparse files, so ftruncate really does go allocate books of storage when increasing the size of the file.

12.1.2. fallocate(int fd, int mode, off_t offset, off_t len)

fallocate is a more flexible and sophisticated way of constructing a file. fallocate is not a part of POSIX, but it is a documented Linux system call. It offers a more flexible way than ftruncate for managing storage of a file

LFS doesn’t support sparse files, or the ability to shuffle blocks around within a file so many of the values for *mode* are not supported and will return EOPNOTSUPP. Those are listed below. We could support some of these operations, but in the absense of a compelling use case, we’re going to restrict the use of this syscall to the equivalent of ftruncate and only let it set the size of the file

FALLOC_FL_KEEP_SIZE

This works the same as it does on other file systems, restricting fallocate to not change the size of the file, which effectively turns it into a no-op as we don’t support any of the extended modes.

FALLOC_FL_PUNCH_HOLE

Not supported. Returns errno of EOPNOTSUPP.

FALLOC_FL_COLLAPSE_RANGE

Not supported. Returns errno of EOPNOTSUPP.

FALLOC_FL_ZERO_RANGE

Not supported. Returns errno of EOPNOTSUPP.

12.1.3. flock(int fd, int operation)

While the library file system is visible across the whole Load/Store Domain (LSD), locks made with flock are local to each node. Making locking work across nodes means preserving state within the librarian and tracking when nodes crash and recover. We’ll avoid this until someone comes up with a compelling use case.

12.1.4. fcntl(int fd, int cmd, ... /* arg */)

There are four relevant areas for this call, Advisory record locking, Open file description locks, Mandatory locking and Leases.

As with flock, the current plan is to limit the effect of all of these operations to the current node and not reflect them into the librarian and thence into other nodes.

12.1.5. fsetxattr(int filedes, const char *name,

```
const void *value, size_t size, int flags)
```

The *user.interleave_request* extended attribute directs the Librarian of the desired location of future FAM allocations for a shelf. It is encoded using one byte for each book position in the shelf, each byte naming the interleave group that a future allocation of a book in that position should be allocated from. This is only a hint to the Librarian; if no FAM can be allocated from the specified interleave group, the Librarian will allocate FAM from another group. For regions of the shelf not described in this attribute (and when the attribute is not set at all), the Librarian is free to allocate books from any interleave group in the system.

The *user.interleave* extended attribute is a read-only attribute which describes where each book in the shelf is actually allocated, in cases where the Librarian was

unable to allocate books in the requested shelves, or when the requested shelf was changed after allocate occurred.

There are also users commands, getfattr and setfattr, which can perform this operation from the command line.

12.1.6. int mkdir(const char *pathname, mode_t mode)

The current LFS implementation doesn’t support sub-directories, so this will return -1 and set errno to ENOTSUP.

12.1.7. int symlink(const char *target, const char *linkpath)

LFS doesn’t support symlinks, so this will return -1 and set errno to ENOTSUP

12.1.8. int link(const char *target, const char *linkpath)

LFS doesn’t support hard links, so this will return -1 and set errno to ENOTSUP

12.2. Memory mapping syscalls and LFS

This section provides details on the semantics of each of the memory-mapping system calls. As LFS maps FAM directly into an application’s virtual address space, some of the semantics of the memory mapping system calls are subtly affected.

One general thing to remember is that until you call pmem_persist, no writes are guaranteed to be saved to FAM. If the SoC crashes, then any unflushed data left in SoC caches will not reach FAM.

12.2.1. mmap

mmap provides for direct CPU access to the contents of a file. In the LFS environment, that is done by directly mapping FAM into the application. There is no buffering between the application and memory, so writes to FAM, once drained with pmem_drain, will be both visible across the entire LSD and persistent across OS crashes or reboots.

Here’s a description of how some of the flags work on LFS:

MAP_SHARED

All writes go directly to FAM addresses. They will be immediately visible to other tasks on the local SoC. Of course, writes will not be visible to tasks on other SoCs until the associated cache lines are written to FAM and the other SoC reads those cache lines from FAM.

MAP_PRIVATE

This will create a Copy on Write (COW) mapping. Writes through this mapping will copy the underlying content into a newly allocated page in direct-attached DRAM. Therefore, writes through these mappings will not be visible through other mappings on the same or other SoCs.

MAP_LOCKED

See the description of mlock below.

MAP_POPULATE

Because the physical aperture on the SoC is far smaller than the size of FAM, MAP_POPULATE may not be able to allocate PA space for the entire mapping. As a result, MAP_POPULATE may not end up avoiding page faults during application operation.

MAP_HUGETLB

Use Huge pages for this mapping. Each TLB entry covers 512MB of address space rather than 64kB, potentially reducing the number of TLB misses.

MAP_ANONYMOUS

This always comes out of system memory or swap (the fd and offset arguments are ignored), so this will not be in FAM.

Other mmap flags don’t have any particularly interesting behaviour when associated with LFS, and should thus operate as documented in the mmap(2) man page.

12.2.2. mlock

mlock is supposed to prevent paging for the specified region. Because the physical aperture for FAM is smaller than FAM, we can’t guarantee that no page faults will occur even if you call mlock. When we do replace mappings within the aperture, the SoC caches will be flushed and invalidated for the affected physical address range, and future access to virtual addresses which have had their physical addresses invalidated will generate page faults.

If we really want mlock to eliminate pagefaults, then we'll have to wire down aperture space for the mlock'd region, and that will have drastic effects on the overall system performance as we will run out of aperture space for other operations far more quickly.

12.2.3. mprotect

We don't know the state of mprotect support in the current DAX implementation, and have no plans to work in this area, so what you see is what you get.

12.2.4. mremap

There aren't any semantic changes for this call

12.2.5. msync

For DAX-mapped files, this will have the same effect as using pmem_persist on the specified virtual address range.

MS_ASYNC

msync may return before FAM is updated.

MS_INVALIDATE

Not relevant for DAX-mapped files; if two processes have mapped the same file through DAX, there aren't any buffers to invalidate.

12.2.6. munmap

munmap is not guaranteed to perform cache flushing on the unmapped range. If you want writes to the unmapped region to become visible to other SoCs, you will want to use pmem_persist before calling munmap.

12.2.7. shm_open

POSIX shared memory objects are not allocated in FAM, rather they are allocated in normal direct-attached memory. Shared memory objects are not visible outside of the local SoC operating system environment.

12.3. Librarian File System Control

The library file system uses the standard POSIX APIs described above for most operations. For operations beyond the standard POSIX semantics, LFS defines a set

of ioctl(2) operations. These operations are wrapped in the librarian_control libary, which is specified in the following man page:

12.3.1. NAME

librarian_control — librarian file system control interface

```
#include <librarian_control.h>

librarian_set_mapping
librarian_clear_mapping
librarian_register_sigbus
librarian_unregister_sigbus
```

12.3.2. SYNOPSIS

Control zBridge mapping granularity

```
int
librarian_set_mapping(int fd, int64_t offset, int64_t len, int mode);

int
librarian_clear_mapping(int fd, int64_t offset, int64_t len);
```

Register SIGBUS handler for memory region

```
int
librarian_register_sigbus(void (*action)(int signo, siginfo_t *info, void *addr));

void
librarian_unregister_sigbus(void (*action)(int signo, siginfo_t *info, void *addr));
```

12.3.3. DESCRIPTION

The Librarian Control library provides interfaces for dealing with the vageries of fabric attached memory. The two current sets of functionality are controlling the Z address to physical address mapping granularity and providing a central location for managing SIGBUS handlers.

12.3.4. MAPPING GRANULARITY CONTROL

The address mapping translation unit offers control over the granularity of zaddr to paddr address translation mappings used when accessing fabric attached memory. Choosing the optimal granularity provides a trade-off between mapping size efficiency and the cost of changing the mapping tables.

Regions of the file not covered by mapping specifications will get a kernel-defined mapping granularity.

librarian_set_mapping(int fd, int64_t offset, int64_t len, int mode) selects the mapping granularity to be used for future mmap calls in the range of the *fd* from

offset to *offset* + *len* - *1*. This replaces any existing mapping mode settings for this portion of the file but does not affect existing mappings within the file. *mode* can be one of:

LIBRARIAN_MAPPING_8GB

Granularity 8 GB.

LIBRARIAN_MAPPING_64KB

Granularity 64 kB.

LIBRARIAN_MAPPING_BOOK

Book granularity (an alias for LIBRARIAN_MAPPING_8GB).

LIBRARIAN_MAPPING_BOOKLET

Booklet granularity (an alias for LIBRARIAN_MAPPING_64KB).

librarian_clear_mapping(int fd, int64_t offset, int64_t len) clears the mapping selection for *fd* in the range *offset* to *offset* + *len* - *1*.

12.3.5. SIGBUS HANDLING

SIGBUS is delivered when an error occurs in Fabric Attached memory mapped by the process. To handle multiple subsystems all wanting to receive SIGBUS notification, a simple registration API is provided.

At SIGBUS time, all registered handlers are invoked with the standard sigaction parameters.

librarian_register_sigbus(void (*action)(int signo, siginfo_t *info, void *addr)) Registers a function to be called at SIGBUS time.

librarian_unregister_sigbus(void (*sigaction)(int signo, siginfo_t *info, void *addr)). Unregister a SIGBUS handler.

12.3.6. RETURN VALUE

librarian_set_mapping and **librarian_get_mapping** return 0 on success and a negative value on failure.

libarian_register_sigbus returns 0 on success and a negative value on failure. **librarian_unregister_sigbus** has no failure modes.

12.4. fam_atomic(3)

12.4.1. NAME

fam_atomic - fabric attached memory atomic support library

12.4.2. SYNOPSIS

Include the header file

```
#include <fam_atomic.h>
```

Register and unregister FAM areas with the library

```
int
fam_atomic_register_region(void *region_start,
                           size_t region_length,
                           int fd,
                           off_t offset);

void
fam_atomic_unregister_region(void *region_start,
                             size_t region_length);
```

Unpadded atomic operations. These are not cacheline aligned, so care must be taken when using them to avoid accessing memory within the same cacheline from the SoC without performing the required cache flush and invalidate operations.

```
int32_t
fam_atomic_32_fetch_add(int32_t *address,
                        int32_t increment);

int64_t
fam_atomic_64_fetch_add(int64_t *address,
                        int64_t increment);

int32_t
fam_atomic_32_swap(int32_t *address,
                   int32_t value);

int64_t
fam_atomic_64_swap(int64_t *address,
                   int64_t value);

void
fam_atomic_128_swap(int64_t *address,
                    int64_t value[2],
                    int64_t result[2]);

int32_t
fam_atomic_32_compare_store(int32_t *address,
                            int32_t compare,
                            int32_t store);
```

```
int64_t
fam_atomic_64_compare_store(int64_t *address,
                            int64_t compare,
                            int64_t store);

void
fam_atomic_128_compare_store(int64_t *address,
                             int64_t compare[2],
                             int64_t store[2],
                             int64_t result[2]);

int32_t
fam_atomic_32_read(int32_t *address);

int64_t
fam_atomic_64_read(int64_t *address);

void
fam_atomic_128_read(int64_t *address,
                    int64_t result[2]);

void
fam_atomic_32_write(int32_t *address,
                    int32_t value);

void
fam_atomic_64_write(int64_t *address,
                    int64_t value);

void
fam_atomic_128_write(int64_t *address,
                     int64_t value[2]);

int32_t
fam_atomic_32_fetch_and(int32_t *address,
                        int32_t arg);

int64_t
fam_atomic_64_fetch_and(int64_t *address,
                        int64_t arg);

int32_t
fam_atomic_32_fetch_or(int32_t *address,
                       int32_t arg);

int64_t
fam_atomic_64_fetch_or(int64_t *address,
                       int64_t arg);
```

Spinlock operations unpadded

```
struct fam_spinlock;

void
fam_spin_lock(struct fam_spinlock *lock);

bool
fam_spin_trylock(struct fam_spinlock *lock);

void
fam_spin_unlock(struct fam_spinlock *lock);
```

12.4.3. DESCRIPTION

The Fabric Attached Memory Atomics library provides a set of primitives similar to the C11 atomics interfaces, but for memory contained within a cache-incoherent fabric environment.

There are four operations available:

1. Fetch and Add. Read from the target address, stores the sum of that value and a user-provided value and returns the original value. Available for 32- and 64- bit values.

2. Swap. Read from the target address, write a user-provided value to that address and return the original value. Available for 32-, 64- and 128- bit values.

3. Compare and Store. Read from the target address, if that matches one user-provided value, store a second user-provided value. In any case, return the original value. Available for 32-, 64-, and 128- bit values.

4. Read. Reads from the target address, returning that value. Available for 128- bit values. The fetch-and-add operation can be used to perform this on 32- and 64-bit values by providing a 0 value.

All of these operations are atomic at the fabric level. None of them perform any cache flushing or invalidation, so applications also accessing the same cache lines directly must perform the necessary cache invalidation and flushing operations.

Applications must take care when using these functions that either all memory within the same cache line is accessed solely through this atomics API or that the cache line is flushed before an atomic operation and invalidated afterwards.

The atomic functions use virtual memory addresses to reference the atomic objects. That address must be converted to an offset within a specific file before being passed to the kernel. To perform this conversion, the library needs to know the relationship between virtual addresses and files. Applications must call fam_atomic_register_region with the values from the mmap operation to set up this association (see the example). When the region is unmapped, fam_atomic_unregister_region must be called to clear this mapping.

For regions of memory mapped from files not contained within fabric attached memory, these atomic operations will be performed using the C11 atomic operations so that applications can use this library without regard to the location of the storage.

To initialize a spinlock, assign the FAM_SPINLOCK_INITIALIZER.

12.4.4. EXAMPLE

Here's a simple example which maps a new shelf in FAM to cover a structure containing a 64-bit atomic and a spinlock. Then it shows how to invoke some of the functions described above while checking to make sure the library returns the expected values.

```
/*
 * Copyright © 2015, Hewlett Packard Enterprise Development LP
 *
 * Author: Keith Packard <packard@hpe.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation, either version 3 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 */

#include <unistd.h>
#include <fcntl.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
#include <fam_atomic.h>

struct data {
        /*
         * Regular fam atomic and fam spinlock.
         * Each takes up an entire cacheline.
         */
        int64_t                 atomic;
        struct fam_spinlock     spinlock;
} __attribute__((__aligned__(64)));

int
main(int argc, char **argv) {
        char *file = "test.dat";
        int fd = open(file, O_CREAT | O_RDWR, 0666);
        unlink(file);
        ftruncate(fd, sizeof(struct data));
        struct data *data = mmap(0, sizeof(struct data),
                                 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

        /*
         * We must register the region with the fam atomic library
         * before the fam atomics and locks within the region can be used.
         */
        if (fam_atomic_register_region(data, sizeof(struct data), fd, 0) == -1) {
                fprintf(stderr, "unable to register atomic region\n");
                return 1;
        }
```

```
        /* Set to zero */
        fam_atomic_64_write(&data->atomic, 0);

        /* Swap with 12, make sure previous was zero */
        int64_t prev = fam_atomic_64_fetch_add(&data->atomic, 12);
        assert(prev == 0);

        /* Make sure value is now 12 */
        int64_t next = fam_atomic_64_read(&data->atomic);
        assert(next == 12);

        /* Initialize spinlock */
        data->spinlock = FAM_SPINLOCK_INITIALIZER;

        /* Lock it */
        fam_spin_lock(&data->spinlock);

        /* Make sure trylock fails now */
        bool trylock_locked = fam_spin_trylock(&data->spinlock);
        assert(!trylock_locked);

        /* Unlock */
        fam_spin_unlock(&data->spinlock);

        /* Make sure trylock succeeds now */
        bool trylock_unlocked = fam_spin_trylock(&data->spinlock);
        assert(trylock_unlocked);

        /* Unlock */
        fam_spin_unlock(&data->spinlock);

        /* Clean up */
        fam_atomic_unregister_region(data, sizeof(struct data));
        return 0;
}
```

12.4.5. fam_atomic_register_region(3)

NAME

fam_atomic_register_region - registers FAM area with the library

SYNOPSIS

```
#include <fam_atomic.h>

int fam_atomic_register_region(void *region_start,
                               size_t region_length,
                               int fd,
                               off_t offset);

cc ... -lfam_atomic
```

DESCRIPTION

Memory regions containing fam atomics and/or fam locks must be registered using some of the parameters from the mmap operation before any of the atomics or locks within the region can be used. It is recommended that this gets called right after the mmap operation to register the entire mapped region.

region_start: Pointer to start of region. This is usually the value that was returned by mmap().

region_length: Size of the region. Should match value given to mmap().

fd: The file descriptor associated with the region. Should match the fd given to mmap().

offset: The offset between the start of the file and *region_start*. Should match offset value given to mmap().

RETURN VALUE

Returns 0 on success. On failure, returns -1 and sets errno to indicate the error.

ERRORS

ENOMEM Not enough memory for the library to allocate the metadata required for tracking the registered region.

12.4.6. fam_atomic_unregister_region(3)

NAME

fam_atomic_unregister_region - unregisters FAM area with the library

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_atomic_unregister_region(void *region_start,
                             size_t region_length);

cc ... -lfam_atomic
```

DESCRIPTION

Unregisters an FAM area. This should be called on a region just before it gets unmapped if the region was previously registered with fam_atomic_register_region().

Applications must unregister the exact region that was registered with fam_atomic_register_region(). The library does not support unregistering subsets or parts of a registered region.

region_start: Pointer to start of region. Should match the value given to fam_atomic_register_region() when the region was registered.

region_length: Size of the region to unregister.

RETURN VALUE

None

12.4.7. fam_atomic_32_fetch_add(3)

NAME

fam_atomic_32_fetch_add - 32 bit fam atomic fetch and add

SYNOPSIS

```
#include <fam_atomic.h>

int32_t
fam_atomic_32_fetch_add(int32_t *address,
                        int32_t increment);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, reads the contents of the atomic variable, adds *increment* to the variable, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before *increment* was added).

12.4.8. fam_atomic_64_fetch_add(3)

NAME

fam_atomic_64_fetch_add - 64 bit fam atomic fetch and add

SYNOPSIS

```
#include <fam_atomic.h>

int64_t
fam_atomic_64_fetch_add(int64_t *address,
                        int64_t increment);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, reads the contents of the atomic variable, adds *increment* to the variable, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before *increment* was added).

12.4.9. fam_atomic_32_swap(3)

NAME

fam_atomic_32_swap - 32 bit fam atomic swap

SYNOPSIS

```
#include <fam_atomic.h>

int32_t
fam_atomic_32_swap(int32_t *address,
                   int32_t value);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, reads the contents of the atomic variable, sets the variable to *value*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before it was set to *value*).

12.4.10. fam_atomic_64_swap(3)

NAME

fam_atomic_64_swap - 64 bit fam atomic swap

SYNOPSIS

```
#include <fam_atomic.h>

int64_t
fam_atomic_64_swap(int64_t *address,
                   int64_t value);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, reads the contents of the atomic variable, sets the variable to *value*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before it was set to *value*).

12.4.11. fam_atomic_128_swap(3)

NAME

fam_atomic_128_swap - 128 bit fam atomic swap

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_atomic_128_swap(int64_t *address,
                    int64_t value[2],
                    int64_t result[2]);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 128 bit fam atomic variable, reads the contents of the atomic variable, sets the variable to *value*, and stores the original value that was read into *result*.

value and *result* are both arrays of two 64 bit integers representing 128 bit variables, where index [0] represents the first 64 bits and index [1] represents the last 64 bits.

RETURN VALUE

None, the previous value gets stored in *result*.

12.4.12. fam_atomic_32_compare_store(3)

NAME

fam_atomic_32_compare_store - 32 bit fam atomic compare and store

SYNOPSIS

```
#include <fam_atomic.h>

int32_t
fam_atomic_32_compare_store(int32_t *address,
                            int32_t compare,
                            int32_t store);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, reads the contents of the atomic variable and checks if it is equal to *compare*. If true, the variable gets set to *store*. If false, the variable does not get modified. Returns the original value that was read.

After calling this function, applications would typically check if the return value is equivalent to *compare* to verify if the operation succeeded.

RETURN VALUE

Returns the previous value of the fam atomic.

12.4.13. fam_atomic_64_compare_store(3)

NAME

fam_atomic_64_compare_store - 64 bit fam atomic compare and store

SYNOPSIS

```
#include <fam_atomic.h>

int64_t
fam_atomic_64_compare_store(int64_t *address,
                            int64_t compare,
                            int64_t store);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, reads the contents of the atomic variable and checks if it is equal to *compare*. If true, the variable gets set to *store*. If false, the variable does not get modified. Returns the original value that was read.

After calling this function, applications would typically check if the return value is equivalent to *compare* to verify if the operation succeeded.

RETURN VALUE

Returns the previous value of the fam atomic.

12.4.14. fam_atomic_128_compare_store(3)

NAME

fam_atomic_128_compare_store - 128 bit fam atomic compare and store

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_atomic_128_compare_store(int64_t *address,
                             int64_t compare[2],
                             int64_t store[2],
                             int64_t result[2]);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 128 bit fam atomic variable, reads the contents of the atomic variable and checks if it is equal to *compare*. If true, the variable gets set to *store*. If false, the variable does not get modified. The original value that was read gets stored into *result*.

compare, *store,* and *result* are all arrays of two 64 bit integers representing 128 bit variables, where index [0] represents the first 64 bits and index [1] represents the last 64 bits.

After calling this function, applications would typically check if *result* is equivalent to *compare* to verify if the operation succeeded.

RETURN VALUE

None, the previous value gets stored in *result*.

12.4.15. fam_atomic_32_read(3)

NAME

fam_atomic_32_read - 32 bit fam atomic read

SYNOPSIS

```
#include <fam_atomic.h>

int32_t
fam_atomic_32_read(int32_t *address);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, reads the contents of the atomic variable and returns that value.

RETURN VALUE

Returns the value of the fam atomic that was read.

12.4.16. fam_atomic_64_read(3)

NAME

fam_atomic_64_read - 64 bit fam atomic read

SYNOPSIS

```
#include <fam_atomic.h>

int64_t
fam_atomic_64_read(int64_t *address);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, reads the contents of the atomic variable and returns that value.

RETURN VALUE

Returns the value of the fam atomic that was read.

12.4.17. fam_atomic_128_read(3)

NAME

fam_atomic_128_read - 128 bit fam atomic read

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_atomic_128_read(int64_t *address,
                    int64_t result[2]);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 128 bit fam atomic variable, reads the contents of the atomic variable and stores that value into *result*.

result is an array of two 64 bit integers representing a 128 bit variable, where index [0] represents the first 64 bits and index [1] represents the last 64 bits.

RETURN VALUE

None, the value read gets stored in *result*.

12.4.18. fam_atomic_32_write(3)

NAME

fam_atomic_32_write - 32 bit fam atomic write

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_atomic_32_write(int32_t *address,
                    int32_t value);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, sets the atomic variable to *value*. This is essentially an atomic swap that does not return a value.

RETURN VALUE

None

12.4.19. fam_atomic_64_write(3)

NAME

fam_atomic_64_write - 64 bit fam atomic write

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_atomic_64_write(int64_t *address,
                    int64_t value);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, sets the atomic variable to *value*. This is essentially an atomic swap that does not return a value.

RETURN VALUE

None

12.4.20. fam_atomic_128_write(3)

NAME

fam_atomic_128_write - 128 bit fam atomic write

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_atomic_128_write(int64_t *address,
                     int64_t value[2]);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 128 bit fam atomic variable, sets the atomic variable to *value*. This is essentially an atomic swap that does not return a value.

value is an array of two 64 bit integers representing a 128 bit variable, where index [0] represents the first 64 bits and index [1] represents the last 64 bits.

RETURN VALUE

None

12.4.21. fam_atomic_32_fetch_and(3)

NAME

fam_atomic_32_fetch_and - 32 bit fam atomic fetch and operation

SYNOPSIS

```
#include <fam_atomic.h>

int32_t
fam_atomic_32_fetch_and(int32_t *address,
                        int32_t arg);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, reads the contents of the atomic variable, performs bitwise AND between the atomic variable and *arg*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before the atomic variable was AND'd with *arg*.

12.4.22. fam_atomic_64_fetch_and(3)

NAME

fam_atomic_64_fetch_and - 64 bit fam atomic fetch and operation

SYNOPSIS

```
#include <fam_atomic.h>

int64_t
fam_atomic_64_fetch_and(int64_t *address,
                        int64_t arg);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, reads the contents of the atomic variable, performs bitwise AND between the atomic variable and *arg*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before the atomic variable was AND'd with *arg*.

12.4.23. fam_atomic_32_fetch_or(3)

NAME

fam_atomic_32_fetch_or - 32 bit fam atomic fetch or operation

SYNOPSIS

```
#include <fam_atomic.h>

int32_t
fam_atomic_32_fetch_or(int32_t *address,
                       int32_t arg);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, reads the contents of the atomic variable, performs bitwise OR between the atomic variable and *arg*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before the atomic variable was OR’d with *arg*.

12.4.24. fam_atomic_64_fetch_or(3)

NAME

fam_atomic_64_fetch_or - 64 bit fam atomic fetch or operation

SYNOPSIS

```
#include <fam_atomic.h>

int64_t
fam_atomic_64_fetch_or(int64_t *address,
                       int64_t arg);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, reads the contents of the atomic variable, performs bitwise OR between the atomic variable and *arg*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before the atomic variable was OR'd with *arg*.

12.4.25. fam_atomic_32_fetch_xor(3)

NAME

fam_atomic_32_fetch_xor - 32 bit fam atomic fetch xor operation

SYNOPSIS

```
#include <fam_atomic.h>

int32_t
fam_atomic_32_fetch_xor(int32_t *address,
                        int32_t arg);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 32 bit fam atomic variable, reads the contents of the atomic variable, performs bitwise XOR between the atomic variable and *arg*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before the atomic variable was XOR'd with *arg*.

12.4.26. fam_atomic_64_fetch_xor(3)

NAME

fam_atomic_64_fetch_xor - 64 bit fam atomic fetch xor operation

SYNOPSIS

```
#include <fam_atomic.h>

int64_t
fam_atomic_64_fetch_xor(int64_t *address,
                        int64_t arg);

cc ... -lfam_atomic
```

DESCRIPTION

Given *address*, which is a pointer to a 64 bit fam atomic variable, reads the contents of the atomic variable, performs bitwise XOR between the atomic variable and *arg*, and returns the original value that was read.

RETURN VALUE

Returns the previous value of the fam atomic (before the atomic variable was XOR’d with *arg*.

12.4.27. fam_spin_lock_init(3)

NAME

fam_spin_lock_init - Initializes an fam spinlock

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_spin_lock_init(struct fam_spinlock *lock);

cc ... -lfam_atomic
```

DESCRIPTION

Given *lock*, which is a pointer to an fam spinlock, initializes the lock. The lock must be initialized before it gets used.

RETURN VALUE

None

12.4.28. fam_spin_lock(3)

NAME

fam_spin_lock - Acquires an fam spinlock

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_spin_lock(struct fam_spinlock *lock);

cc ... -lfam_atomic
```

DESCRIPTION

Given *lock*, which is a pointer to an fam spinlock, tries to acquire the lock. If the lock has already been acquired by another thread, the current thread will continue to spin until it has acquired the lock.

Used for obtaining mutual exclusion. Only 1 thread could obtain the lock at a time.

RETURN VALUE

None

12.4.29. fam_spin_unlock(3)

NAME

fam_spin_unlock - Releases an fam spinlock

SYNOPSIS

```
#include <fam_atomic.h>

void
fam_spin_unlock(struct fam_spinlock *lock);

cc ... -lfam_atomic
```

DESCRIPTION

Given *lock*, which is a pointer to an fam spinlock, releases the lock that was previously acquired from fam_spin_lock(lock) or a successful fam_spin_trylock(lock).

RETURN VALUE

None

12.4.30. fam_spin_trylock(3)

NAME

fam_spin_trylock - Tries once to acquire an fam spinlock.

SYNOPSIS

```
#include <fam_atomic.h>

bool
fam_spin_trylock(struct fam_spinlock *lock);

cc ... -lfam_atomic
```

DESCRIPTION

Given *lock*, which is a pointer to an fam spinlock, tries once to acquire the lock. If that is successful, the function will return true. If the lock could not be immediately obtained, the trylock will return with false. No spinning occurs in this function.

RETURN VALUE

Returns true if the lock has been acquired, else returns false.

12.5. pmem(3)

12.5.1. NAME

libpmem — persistent memory support library

12.5.2. SYNOPSIS

```
#include <libpmem.h>

cc ... -lpmem
```

Most commonly used functions:

```
int pmem_is_pmem(void *addr, size_t len);
void pmem_persist(void *addr, size_t len);
int pmem_msync(void *addr, size_t len);
void *pmem_map(int fd);
```

Partial flushing operations:

```
void pmem_flush(void *addr, size_t len);
void pmem_drain(void);
int pmem_has_hw_drain(void);
```

Additions for non cache-coherent environments

```
void pmem_invalidate(void *addr, size_t len);
int pmem_needs_invalidate(void *addr, size_t len);
```

Copying to persistent memory:

```
void *pmem_memmove_persist(void *pmemdest, const void *src, size_t len);
void *pmem_memcpy_persist(void *pmemdest, const void *src, size_t len);
void *pmem_memset_persist(void *pmemdest, int c, size_t len);
void *pmem_memmove_nodrain(void *pmemdest, const void *src, size_t len);
void *pmem_memcpy_nodrain(void *pmemdest, const void *src, size_t len);
void *pmem_memset_nodrain(void *pmemdest, int c, size_t len);
```

Library API versioning:

```
const char *pmem_check_version(
    unsigned major_required,
    unsigned minor_required);
```

Error handling:

```
const char *pmem_errormsg(void);
```

12.5.3. DESCRIPTION

libpmem provides low-level persistent memory (pmem) support for applications using direct access storage (DAX), which is storage that supports load/store access without paging blocks from a block storage device. Some types of non-volatile memory DIMMs (NVDIMMs) provide this type of byte addressable access to storage.

A persistent memory aware file system is typically used to expose the direct access to applications. Memory mapping a file from this type of file system results in load/store, non-paged access to pmem.

This library is for applications that use persistent memory directly, without the help of any library-supplied transactions or memory allocation. Higher-level libraries that build on libpmem are recommended for most applications.

Under normal usage, libpmem will never print messages or intentionally cause the process to exit. The only exception to this is the debugging information, when enabled, as described under DEBUGGING AND ERROR HANDLING below.

In The Machine, when a memory error (read or write) is detected, the processes affected will be sent a SIGBUS signal, allowing the process to attempt to recover. As writes are not immediately sent to memory, memory write errors cannot be detected synchronously, and are instead noticed when the cache line is flushed.

pmem_drain is a memory synchronization point beyond which writes cannot be delayed, offering an ideal opportunity to catch and report errors. Any memory error caused by a write occurring before pmem_drain is called will be detected and the application notified via SIGBUS before pmem_drain returns.

pmem_invalidate removes references to the specified range from the CPU cache ensuring that further reads from within that range will result in fetching the cacheline again. pmem_invalidate is needed whenever the same physical memory is mapped by two or more processors in different cache coherence domains.

pmem_needs_invalidate returns TRUE (non-zero) for address ranges which may be accessible from more than one thread or processor in a cache-incoherent fashion. It returns FALSE (zero) for address ranges for which all accesses are cache coherent. In other words, if this function returns TRUE, then the application must use pmem_invalidate to reliably retrieve modifications to memory made by other threads or processors.

12.5.4. MOST COMMONLY USED FUNCTIONS

Most pmem-aware applications will take advantage of higher level libraries that alleviate the application from calling into libpmem directly. Application developers that wish to access raw memory mapped persistence directly (via mmap(2)) and that wish to take on the responsibility for flushing stores to persistence will find the functions described in this section to be the most commonly used.

```
int pmem_is_pmem(void *addr, size_t len);
```

The pmem_is_pmem() function returns true only if the entire range [addr, addr+len) consists of persistent memory. A true return from pmem_is_pmem() means it is safe to use pmem_persist() and the related functions below to make changes durable for that memory range.

The implementation of pmem_is_pmem() requires a non-trivial amount of work to determine if the given range is entirely persistent memory. For this reason, it is better to call pmem_is_pmem() once when a range of memory is first encountered, save the result, and use the saved result to determine whether pmem_persist() or msync(2) is appropriate for flushing changes to persistence. Calling pmem_is_pmem() each time changes are flushed to persistence will not perform well.

Warning

Using pmem_persist() on a range where pmem_is_pmem() returns false may not do anything useful — use msync(2) instead.

```
void pmem_persist(void *addr, size_t len);
```

Force any changes in the range [addr, addr+len) to be stored durably in persistent memory. This is equivalent to calling msync(2) but may be more optimal and will avoid calling into the kernel if possible. There are no alignment restrictions on the range described by addr and len, but pmem_persist() may expand the range as necessary to meet platform alignment requirements.

Warning

Like msync(2), there is nothing atomic or transactional about this call. Any unwritten stores in the given range will be written, but some stores may have already been written by virtue of normal cache eviction/replacement policies. Correctly written code must not depend on stores waiting until pmem_persist() is called to become persistent — they can become persistent at any time before pmem_persist() is called.

```
int pmem_msync(void *addr, size_t len);
```

The function pmem_msync() is like pmem_persist() in that it forces any changes in the range [addr, addr+len) to be stored durably. Since it calls msync(), this function works on either persistent memory or a memory mapped file on traditional storage. pmem_msync() takes steps to ensure the alignment of addresses and lengths passed to msync() meet the requirements of that system call. It calls msync() with the MS_SYNC flag as described in msync(2). Typically the application only checks for the existence of persistent memory once, and then uses that result throughout the program, for example:

```
/* do this call once, after the pmem is memory mapped */
int is_pmem = pmem_is_pmem(rangeaddr, rangelen);

/* ... make changes to a range of pmem ... */

/* make the changes durable */
if (is_pmem)
        pmem_persist(subrangeaddr, subrangelen);
else
        pmem_msync(subrangeaddr, subrangelen);
/* ... */
```

The return value of pmem_msync() is the return value of msync(), which can return -1 and set errno to indicate an error.

```
void *pmem_map(int fd);
```

The pmem_map() function creates a new read/write mapping for the entire file referred by to file descriptor fd, where fd must be a file descriptor for a file opened for both reading and writing. pmem_map() will map the file using mmap(2), but it also takes extra steps to make large page mappings more likely. On success, pmem_map() returns a pointer to mapped area. On error, NULL is returned, and errno is set appropriately. To delete mappings created with pmem_map(), use munmap(2).

The mapping portion of this call is equivalent to:

```
mmap(NULL, file_length, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

12.5.5. PARTIAL FLUSHING OPERATIONS

The functions in this section provide access to the stages of flushing to persistence, for the less common cases where an application needs more control of the flushing operations than the pmem_persist() function described above.

```
void pmem_flush(void *addr, size_t len);
void pmem_drain(void);
```

These functions provide partial versions of the pmem_persist() function described above. pmem_persist() can be thought of as this:

```
void
pmem_persist(void *addr, size_t len)
{
    /* flush the processor caches */
    pmem_flush(addr, len);

    /* wait for any pmem stores to drain from HW buffers */
    pmem_drain();
}
```

These functions allow advanced programs to create their own variations of pmem_persist(). For example, a program that needs to flush several discontiguous ranges can call pmem_flush() for each range and then follow up by calling pmem_drain() once.

Note

Some software is designed for custom platforms that obviate the need for using PCOMMIT (perhaps the platform issues PCOMMIT on shutdown or something similar). Even in such cases, it is recommended that applications using libpmem do not skip the step of calling pmem_drain(), either directly or by using pmem_persist(). The recommended way to inhibit use of the PCOMMIT instruction is by setting the PMEM_NO_PCOMMIT environment variable as described in the ENVIRONMENT VARIABLES section.

```
int pmem_has_hw_drain(void);
```

The pmem_has_hw_drain() function returns if the machine supports the hardware drain function for persistent memory, such as that provided by the PCOMMIT instruction on Intel processors. If support for hardware drain is not found, or cannot be detected by the library, pmem_has_hw_drain() will return false. Although it is typically an administrative task to provide the correct platform configuration for persistent memory, this function is provided for the less common cases where an application needs to ensure this feature is available. Note that the lack of this feature means that calling pmem_persist() may not fully ensure stores are durable, without additional platform features such as Asynchronous DRAM Refresh (ADR) or something similar.

12.5.6. COPYING TO PERSISTENT MEMORY

The functions in this section provide optimized copying to persistent memory.

```
void *pmem_memmove_persist(void *pmemdest, const void *src, size_t len);
void *pmem_memcpy_persist(void *pmemdest, const void *src, size_t len);
void *pmem_memset_persist(void *pmemdest, int c, size_t len);
```

The pmem_memmove_persist(), pmem_memcpy_persist(), and pmem_memset_persist(), functions provide the same memory copying as their namesakes memmove(3) memcpy(3), and memset(3), and ensure that the result has been flushed to persistence before returning. For example, the following code is functionally equivalent to pmem_memmove_persist():

```
void *
pmem_memmove_persist(void *pmemdest, const void *src, size_t len)
{
        void *retval = memmove(pmemdest, src, len);

        pmem_persist(pmemdest, len);

        return retval;
}
```

Calling pmem_memmove_persist() may out-perform the above code, however, since the libpmem implementation may take advantage of the fact that pmemdest is persistent memory and use instructions such as non-temporal stores to avoid the need to flush processor caches.

Warning

Using these functions where pmem_is_pmem() returns false may not do anything useful. Use the normal libc functions in that case.

```
void *pmem_memmove_nodrain(void *pmemdest, const void *src, size_t len);
void *pmem_memcpy_nodrain(void *pmemdest, const void *src, size_t len);
void *pmem_memset_nodrain(void *pmemdest, int c, size_t len);
```

The pmem_memmove_nodrain(), pmem_memcpy_nodrain() and pmem_memset_nodrain() functions are similar to pmem_memmove_persist(),

pmem_memcpy_persist(), and pmem_memset_persist() described above, except they skip the final pmem_drain() step. This allows applications to optimize cases where several ranges are being copied to persistent memory, followed by a single call to pmem_drain(). The following example illustrates how these functions might be used to avoid multiple calls to pmem_drain() when copying several ranges of memory to pmem:

```
/* ... write several ranges to pmem ... */
pmem_memcpy_nodrain(pmemdest1, src1, len1);
pmem_memcpy_nodrain(pmemdest2, src2, len2);

/* ... */

/* wait for any pmem stores to drain from HW buffers */
pmem_drain();
```

Warning

Using pmem_memmove_nodrain(), pmem_memcpy_nodrain() or pmem_memset_nodrain() on a destination where pmem_is_pmem() returns false may not do anything useful.

12.5.7. LIBRARY API VERSIONING

This section describes how the library API is versioned, allowing applications to work with an evolving API.

```
const char *pmem_check_version(unsigned major_required,
                               unsigned minor_required);
```

The pmem_check_version() function is used to see if the installed libpmem supports the version of the library API required by an application. The easiest way to do this is for the application to supply the compile-time version information, supplied by defines in <libpmem.h>, like this:

```
reason = pmem_check_version(PMEM_MAJOR_VERSION,
                            PMEM_MINOR_VERSION);
if (reason != NULL) {
    /*  version check failed, reason string tells you why */
}
```

Any mismatch in the major version number is considered a failure, but a library with a newer minor version number will pass this check since increasing minor versions imply backwards compatibility.

An application can also check specifically for the existence of an interface by checking for the version where that interface was introduced. These versions are documented in this man page as follows: unless otherwise specified, all interfaces described here are available in version 1.0 of the library. Interfaces added after version 1.0 will contain the text introduced in version x.y in the section of this manual describing the feature.

When the version check performed by pmem_check_version() is successful, the return value is NULL. Otherwise the return value is a static string describing the reason for failing the version check. The string returned by pmem_check_version() must not be modified or freed.

12.5.8. DEBUGGING AND ERROR HANDLING

Two versions of libpmem are typically available on a development system. The normal version, accessed when a program is linked using the -lpmem option, is optimized for performance. That version skips checks that impact performance and never logs any trace information or performs any run-time assertions. If an error is detected during the call to libpmem function, an application may retrieve an error message describing the reason of failure using the following function:

```
const char *pmem_errormsg(void);
```

The pmem_errormsg() function returns a pointer to a static buffer containing the last error message logged for current thread. The error message may include description of the corresponding error code (if errno was set), as returned by strerror(3). The error message buffer is thread-local; errors encountered in one thread do not affect its value in other threads. The buffer is never cleared by any library function; its content is significant only when the return value of the immediately preceding call to libpmem function indicated an error, or if errno was set. The application must not modify or free the error message string, but it may be modified by subsequent calls to other library functions.

A second version of libpmem, accessed when a program uses the libraries under /usr/lib/nvml_debug, contains run-time assertions and trace points. The typical way to access the debug version is to set the environment variable LD_LIBRARY_PATH to /usr/lib/nvml_debug or /usr/lib64/nvml_debug depending on where the debug libraries are installed on the system. The trace points in the debug version of the library are enabled using the environment variable PMEM_LOG_LEVEL, which can be set to the following values:

0

This is the default level when PMEM_LOG_LEVEL is not set. No log messages are emitted at this level.

1

Additional details on any errors detected are logged (in addition to returning the errno-based errors as usual). The same information may be retrieved using pmem_errormsg().

2

A trace of basic operations is logged.

3

This level enables a very verbose amount of function call tracing in the library.

4

This level enables voluminous and fairly obscure tracing information that is likely only useful to the libpmem developers.

The environment variable PMEM_LOG_FILE specifies a file name where all logging information should be written. If the last character in the name is "-", the PID of the current process will be appended to the file name when the log file is created. If PMEM_LOG_FILE is not set, the logging output goes to stderr.

Setting the environment variable PMEM_LOG_LEVEL has no effect on the non-debug version of libpmem.

12.5.9. ENVIRONMENT VARIABLES

Many of these environment variables are only relevant on the x86 version of libpmem. Precisely how these (and potentially other new variables) will effect the arm64 version will be decided during the arm64 porting process

libpmem can change its default behavior based on the following environment variables. These are largely intended for testing and are not normally required.

These are the environment variables available on the x86-64 implementation of libpmem. The set of variables and their behavior on other architectures is not defined at this time.

PMEM_IS_PMEM_FORCE=val

If val is 0 (zero), then pmem_is_pmem() will always return false. Setting val to 1 causes pmem_is_pmem() to always return true. This variable is mostly used for testing but can be used to force pmem behavior on a system where a range of pmem is not detectable as pmem for some reason.

PMEM_NO_PCOMMIT=1

Setting this environment variable to 1 forces libpmem to never issue the Intel PCOMMIT instruction. This can be used on platforms where the hardware drain function is performed some other way, like automatic flushing during a power failure.

Warning

Using this environment variable incorrectly may impact program correctness.

PMEM_NO_CLWB=1

Setting this environment variable to 1 forces libpmem to never issue the CLWB instruction on Intel hardware, falling back to other cache flush instructions instead (CLFLUSHOPT or CLFLUSH on Intel hardware). Without this environment variable, libpmem will always use the CLWB instruction for flushing processor caches on platforms that support the instruction. This variable is intended for use during library testing but may be required for some rare cases where using CLWB has a negative impact on performance.

PMEM_NO_CLFLUSHOPT=1

Setting this environment variable to 1 forces libpmem to never issue the CLFLUSHOPT instruction on Intel hardware, falling back to the CLFLUSH instructions instead. Without this environment variable, libpmem will always use the CLFLUSHOPT instruction for flushing processor caches on platforms that support the instruction, but where CLWB is not available. This variable is intended for use during library testing.

PMEM_NO_MOVNT=1

Setting this environment variable to 1 forces libpmem to never use the non-temporal move instructions on Intel hardware. Without this environment variable, libpmem will use the non-temporal instructions for copying larger ranges to persistent memory on platforms that support the instructions. This variable is intended for use during library testing.

PMEM_MOVNT_THRESHOLD=val

This environment variable allows overriding the minimal length of pmem_memcpy_*(), pmem_memmove_*() or pmem_memset_*() operations, for which libpmem uses non-temporal move instructions. Setting this environment variable to 0 forces libpmem to always use the non-temporal move instructions if available. It has no effect if PMEM_NO_MOVNT variable is set to 1. This variable is intended for use during library testing.

12.5.10. EXAMPLES

The following example uses libpmem to flush changes made to raw, memory-mapped persistent memory.

Warning

there is nothing transactional about the pmem_persist() or pmem_msync() calls in this example. Interrupting the program may result in a partial write to pmem. Use a transactional library to avoid torn updates.

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <libpmem.h>

/* using 4K of pmem for this example */
#define PMEM_LEN 4096

int
main(int argc, char *argv[])
{
    int fd;
    char *pmemaddr;
    int is_pmem;

   /* create a pmem file */
    if ((fd = open("/pmem-fs/myfile",
                        O_CREAT|O_RDWR, 0666)) < 0) {
        perror("open");
        exit(1);
    }

   /* allocate the pmem */
    if ((errno = posix_fallocate(fd, 0, PMEM_LEN)) != 0) {
        perror("posix_fallocate");
        exit(1);
    }

   /* memory map it */
    if ((pmemaddr = pmem_map(fd)) == NULL) {
        perror("pmem_map");
        exit(1);
    }
    close(fd);

   /* determine if range is true pmem */
    is_pmem = pmem_is_pmem(pmemaddr, PMEM_LEN);

   /* store a string to the persistent memory */
    strcpy(pmemaddr, "hello, persistent memory");

   /* flush above strcpy to persistence */
    if (is_pmem)
        pmem_persist(pmemaddr, PMEM_LEN);
    else
        pmem_msync(pmemaddr, PMEM_LEN);
}
```

See http://pmem.io/nvml/libpmem for more examples using the libpmem API.

12.5.11. ACKNOWLEDGMENTS

libpmem builds on the persistent memory programming model recommended by the SNIA NVM Programming Technical Work Group:

http://snia.org/nvmp

12.5.12. SEE ALSO

mmap(2), munmap(2), msync(2), strerror(3), and http://pmem.io.

12.6. RVMA(3)

12.6.1. NAME

RVMA - Remote Virtual Memory Access messaging API for The Machine

12.6.2. DESCRIPTION

The librvma library allows the programmer to do asynchronous messaging between process pairs. The goal is to provide an API that provides access to the best available transport layers, transparently, including the evolving data transport capabilities of the memory fabric. While conceptually similar to RDMA, RVMA does away with memory registration and specifies the memory addresses for transfers by using the virtual addresses of processes. The API is simpler than Infiniband Verbs and, unlike APIs such as MPI, communication is not limited to a fixed set of processes.

The original concept was that a custom RVMA engine built into the memory fabric would provide latency equal to Infiniband with greater bandwidth. The RVMA engine quickly fell out of scope for the June, 2016, but is in-plan for later Machine implementations.

Note

1. The best possible transport between SOCs for the June, 2016, will probably be TCP/IP over 25Gb Ethernet. The RDMA over Converged Ethernet (RoCE) NIC will become available at a later date.
2. Linking to librvma.so will link a program to libpthread, since the library needs threads internally.
3. The current thinking is that RVMA is not primarily an end-programmer API, but an API for porting middleware to the machine.
4. Software that messages using RVMA will transparently take advantage of new transports as they are implemented.

12.6.3. OVERVIEW

Note

1. A simple, complete, example program is shown at EXAMPLE. Having that available may make this manpage easier to understand.
2. Unless otherwise stated, functions return a completion status; if it is negative an error has occurred and value will be a negative system *errno* or an RVMA specific error code. (See rvma_strerror(3))

The most important objects used by the RVMA API are *contexts*, *listeners*, and *events*. These are instantiated only by the API and are seen by the program as pointers --- **struct rvma_ctxt** *, **struct rvma_listener** *, **struct rvma_event** * --- to opaque objects. Contexts represent the association between a pair of processes; listeners represent a process offering a service to other processes; and events are used for synchronization. The term context is used instead of the term connection to try to distance the API from the semantics of a network connection, such as ordered delivery.

RVMA uses a client-server model similar in semantics to that of sockets over TCP/IP. A server process creates a listener for a service using rvma_listener_create(3) and clients connect to the service using rvma_ctxt_create(3). The client specifies a server by hostname or IP; a service name; and a "blob" of service-specific data known as the root data. The server receives the newly created context and can examine the remote host address and the root data provided to determine whether to accept or destroy the context; if it accepts the context with rvma_listener_ctxt_accept(3) --- sending its own root data to the client --- the client call will complete successfully; if the server destroys the context with rvma_ctxt_destroy(3), the client will receive an error. Once created, the context is symmetric, meaning that there is no difference in the API between the client and server processes; the process executing the RVMA call is the local process and the process it acts on is the remote process, which is specified to the API by passing the context.

Note

1. Both rvma_ctxt_create(3) and rvma_listener_create(3) have a parameter *auth* that is a placeholder for a future authentication mechanism; this mechanism will not be implemented for the June, 2016, and *auth* must be NULL.

2. The following access functions exist for contexts:

   a. rvma_ctxt_event(3) returns a pointer to an event created in the local process as part of the context; the address of this event is sent to the remote process as part of the root data it receives.

   b. rvma_ctxt_remote_root(3) provides a pointer to a **struct rvma_root** that contains a pointer to a copy of the root data sent by the remote process and pointer to the context’s event created by the other process.

3. The following access function exists for listeners:

   a. rvma_listener_event(3) provides access to the event the listener uses to wait for new requests; the only reason to access this is if the program needs to wait for multiple events in a single thread, otherwise, rvma_listener_ctxt_get(3) can be used to do the wait.

The root data allows programs using the API to bootstrap their communication. In order to use RVMA, a local process has to know at least one legal address in the remote process on which to perform a put or get; the root data exists to provide that first address; what happens then depends on the programs themselves. The following code snippet from the example shows how the buffer address on the server node was sent to the client via the root data:

```
struct svr_root {
    void  *buf;
};

static int do_server(void)
{
    int buf[2] = { 0, 0 };          /* Data buffer */
    struct svr_root sroot;          /* Root data passed to client. */
    struct rvma_event *sevent;
    ...
    /*
     * sroot will be the root data passed to the client with the address
     * of buf. sevent points to the context's event in this process and
     * will be implicitly passed with the root data to the client.
     */
    sroot.buf = buf;
    sevent = rvma_ctxt_event(ctxt);
    /* Accept context, pass access and sroot. */
    rc = rvma_listener_ctxt_accept(ctxt, access, &sroot, sizeof(sroot));
    ...
}

static int do_client(const char *host)
{
    /* Client, create context. */
    struct rvma_root *rroot;        /* Remote root data from RVMA. */
    struct rvma_event *sevent;      /* Event server will wait on. */
    struct svr_root *sroot;         /* Pointer root data passed by server. */
    ...
    /* Create context between client and server, synchronous version, */
    rc = rvma_ctxt_create(host, SVC_NAME, NULL, access, NULL, 0, &ctxt);
    CHECK_ERROR(rc);
    /* Context fully created, get pointers to server data. */
    rroot = rvma_ctxt_remote_root(ctxt);
    sevent = rroot->event; /* Same as sevent in the server snippet */
    sroot = rroot->addr;
    sbuf = sroot->buf;     /* Same as buf in the server snippet */
    ...
}
```

Events can be created implicitly as part of a context or a listener, or events can be created explicitly by rvma_event_alloc(3). An event has an address in the virtual address space of the process that created it. Events are signaled; tested, and waited-on. Only the process that created it, can wait on an event; but the creating process and any process that has a valid context to the creating process can signal the event. Each time an event is signaled, the pending signal count of the event is incremented. A thread in a process that wants to wait for a signal to an event can call rvma_event_wait(3) --- or its other variants --- and specify the number of signals to wait for and, when they are available, the function acknowledges (and

consumes) the signals atomically and wakes up. Events are used by the API to signal completion of asynchronous operations (see ASYNCHRONOUS OPERATIONS) and by rvma_get(3) and rvma_put(3) to signal buffers are free for reuse. A program may also use events for any kind general notification with rvma_event_signal(3).

Note

1. Multi-event waits exist, see rvma_event_waitall_timeout(3) and rvma_event_waitany_timeout(3)
2. Events have a visible field, *user*. See rvma_event_waitany_timeout(3) for use cases.

Data transfer in RVMA is done with one-sided operations. The local process puts data to the remote process with rvma_put(3) and gets data from the remote process with rvma_get(3). For example:

```
ret = rvma_put(ctxt, rdst, lsrc, len, source_free, rdone_event);
```

This is the synchronous version of rvma_put(3). When the call returns successfully, the operation is complete: all data has been transferred successfully and all signals have been delivered. The destination remote location for the memory is specified as a pair of arguments: *ctxt* and *rdst*. *ctxt* is the context and *rdst* is the virtual address in the process in the remote process; if *ctxt* is NULL, the local process is targeted. The source local address is *lsrc*. *len* is the number of bytes to transfer. *source_free* is an optional local event that will be signaled when all the data from the local source buffer is in-flight and the buffer may be reused; this could allow for better overlap of computation and I/O. *rdone_event* is a optional remote event in the process designated by *ctxt* that can be used to tell that process that the operation completed successfully; with some transports this signal can be piggybacked with the data transfer, which will reduce latency for the signal. Note that if *len* is zero, *source_free* and *rdone_event* will be signaled, if they were specified.

```
ret = rvma_get(ldst, ctxt, rsrc, len, source_free);
```

rvma_get(3) is similar to rvma_put(3), except the remote process is the source of the data to be transferred; *source_free* is an optional remote event that will be signaled when all the data in the remote source buffer is in flight; and there is no *rdone_event*.

Note

1. Puts and gets are subject to access controls. See ACCESS CONTROLS.

Two data transfer operations are temporally ordered if due to the use of synchronous put or get, or through waiting for events, it can be shown that one must complete before the other starts. Any two data transfers that are not temporally ordered are said to be concurrent. If a pair of concurrent transfers uses source or destination buffers that overlap, and at least one of these overlapping buffers is a destination buffer, then that constitutes a data race. The semantics

of a program that contains an RVMA data race are undefined. Moreover, a data transfer is concurrent with itself, again, the semantics are undefined if its source and destination buffers overlap.

After a program is finished with a context, listener, or event, they must be properly freed. For single-threaded programs, this is straightforward. For contexts, the process pair creates their shared contexts; exchange data; and then processes call rvma_ctxt_close(3) to perform and orderly shutdown of a context: all outstanding operations will be allowed to complete; and the context is freed. If one process or another detects an error condition, it can use rvma_ctxt_destroy(3) to unilaterally destroy and free the context: the low-level transport will be closed and outstanding operations will return with error. When and how the remote process detects that the connection has been destroyed depends on the underlying transport: if the network is functioning properly, both the TCP and RDMA implementations will receive notification of the connection being closed and mark their contexts as destroyed. Once the destroy or close functions return, the local process is guaranteed that the remote process cannot affect it using that context. Listeners are simply destroyed with rvma_listener_destroy(3) when a process no longer needs to accept new contexts. Events that were created with rvma_event_alloc(3) are destroyed with rvma_event_destroy(3); however, the events returned by rvma_ctxt_event(3) and rvma_listener_event(3) are destroyed when the corresponding context and listener are destroyed.

For multi-threaded programs, care is required to ensure that API objects are not freed while still in use. Since the RVMA library is internally multi-threaded, it uses reference counts on objects and APIs expose them to the programmer: rvma_xxx_inc_ref/rvma_xxx_dec_ref(3). API objects are created with a reference count of one; when the reference count reaches zero, the object will be freed. For each additional thread that will access an object, an object’s reference count should be incremented by one, which should be decremented only when the thread is through accessing the object.

Note

1. rvma_ctxt_destroy(3), rvma_event_destroy(3), and rvma_listener_destroy(3) decrement the object’s reference count, but they also mark the object as destroyed: calling most APIs with a destroyed object will return an error.

2. rvma_ctxt_close(3) decrements the reference count and marks the object as closed: calling most APIs with a closed context will return an error. If the program has closed the context, no other thread should be trying to reference it, anyway, with the exception of dropping reference counts on other threads.

3. If the program terminates abnormally, it is currently the responsibility of the underlying transport to detect the failure and notify the remote process. The associated context and its internal event will be marked destroyed,

but this may be insufficient to clean up everything, if events other than the context's event are being used.

12.6.4. ASYNCHRONOUS OPERATIONS

Asynchronous functions name's end in ***async*. All such functions have the final arguments _status** and *done_event*. There are two distinct ways these can be used to check operation completion and status. First, the address of variable to hold the status and a completion event can be passed to the routines.

```
...
done_event = rvma_event_alloc();
if (!done_event)
    goto error;
status = 0; // status must be initialized to zero before first async
rvma_put_async(ctxt, &sbuf[0], &buf[0], sizeof(sbuf[0]), NULL, NULL,
               &status, done_event);
rvma_put_async(ctxt, &sbuf[1], &buf[1], sizeof(sbuf[1]), NULL, NULL,
               &status, done_event);
/* Wait for 2 signals => both operations completed. */
ret = rvma_event_wait(done_event, 2);
if (ret < 0 && status < 0)
    goto error;
...
```

In the above example, *status* is updated atomically only if an error occurs and only if status >= 0. More simply, *status* will contain the first error that occurs, if any.

The other way of waiting for asynchronous routines are wait epochs: if both *status* and *done_event* are NULL, an implicit wait object called an epoch is created and shared with each operation with NULL pointers for *status* and *done_event*. Calling rvma_epoch_wait_timeout(3) will wait for all currently outstanding operations and a new epoch will be created for new operations. New epochs can also be explicitly created by rvma_epoch_new(3).

```
...
rvma_put_async(ctxt, &sbuf[0], &buf[0], sizeof(sbuf[0]), NULL, NULL,
               NULL, NULL);
rvma_put_async(ctxt, &sbuf[1], &buf[1], sizeof(sbuf[1]), NULL, NULL,
               NULL, NULL);
if (((ret = rvma_epoch_new() < 0)
    goto error;
rvma_put_async(ctxt, &sbuf[2], &buf[2], sizeof(sbuf[2]), NULL, NULL,
               NULL, NULL);
/* Wait for first two operations to complete. */
if (rvma_epoch_wait_timeout(-1) < 0)
   goto error;
/* Wait for third operation to complete. */
if (rvma_epoch_wait_timeout(-1) < 0)
   goto error;
...
```

Epochs are implemented as a per-context list of epochs with outstanding operations; rvma_epoch_wait_timeout(3) removes the oldest epoch from the tail of the list and waits for all the operations associated with the epoch to complete. The

value it returns will be greater than or equal to zero if no error occurred, or the first error seen.

Finally, for asynchronous operations, the order of issuance does not guarantee the order of execution. The only way to guarantee order of execution is to wait for their completion before issuing new operations. For example:

```
...
done_event = rvma_event_alloc();
if (!done_event)
    goto error;
status = 0; // status must be initialized to zero before first async
rvma_put_async(ctxt, &sbuf[0], &buf[0], sizeof(sbuf[0]), NULL, NULL,
               &status, done_event);
/* Need to guarantee the order of delivery. */
ret = rvma_event_wait(done_event, 1);
if (ret < 0 && status < 0)
    goto error;
rvma_put_async(ctxt, &sbuf[1], &buf[1], sizeof(sbuf[1]), NULL, NULL,
               &status, done_event);
...
```

In order to guarantee data was delivered to *sbuf[0]* before data was delivered to *sbuf[1]*, the rvma_event_wait(3) was required.

12.6.5. A PUT'S A READ (LOCALLY) AND A GET'S A WRITE

A **rvma_put()** is a read of the source buffer by the local thread; the same memory ordering and synchronization that apply to normal reads apply to this as well. In particular, if the data to be sent were to be created by another thread, then the local thread must synchronize with that thread to ensure that the data is visible to the local thread. Conversely, **rvma_get()** is a write by the local thread to the destination buffer in the local process; and the same memory ordering and synchronization that apply to normal writes apply to this.

In the remote process, a **rvma_put()** is a write and a **rvma_get()** is a read. Any thread wishing to synchronize wishing to wait for the completion must wait for a done signal with a RVMA event; that thread can then synchronize with other threads with functions such as **pthread_cond_signal()** and **pthread_cond_wait()**.

12.6.6. FLUSHING FOR PERSISTENCE

Although the FAM for June, 2016, will not actually be persistent memory, programmers should treat it as if it were and have RVMA persist the written data to the FAM, when necessary for correctness. Having the RVMA API query the kernel on every transfer to check whether data should be persisted seems prohibitively expensive, so the programmer must tell the API when to persist data. The rvma_getx(3) and rvma_putx(3) functions add a *flags* argument that can be used to request persisting the data at the destination.

12.6.7. ACCESS CONTROLS

rvma_ctxt_create(3) and rvma_listener_ctxt_accept(3) accept the parameter *access*, which is used to specify a set of memory ranges and access permissions for RVMA operations. The access list **struct rvma_access** * is built up though multiple calls to rvma_access_add_range(3). The access list must be stable until the context is created and the list is copied into the context; after which, changes to the access list will not affect the context and the access list should be freed with rvma_access_free(3). After a context is created, new access ranges can be added with rvma_ctxt_access_add_range(3).

```
...
int rc;
const size_t buf_len = 80;
struct rvma_access *access = NULL;
char *buf;

buf = malloc(buf_len);
if (!buf)
    goto error;
/* Make buf[] the only memory that can be accessed. */
rc = rvma_access_add_range(&access, buf, buf_len,
                           RVMA_ACCESS_RD | RVMA_ACCESS_WR);
if (rc < 0)
    goto error;
rc = rvma_ctxt_create(host, SVC_NAME, NULL, access, NULL, 0, &ctxt);
/* The access list needs to be freed. */
rvma_access_free(access);
```

12.6.8. ATOMIC OPERATIONS

rvma_atomic_fetch_op(3) implements atomic operations for RVMA. Notionally, these operations are executed by shipping the argument data to the remote process and executing the standard gcc atomic builtin functions inside that process.

Note

1. The RVMA atomic operations are incompatible with the FAM atomic variables provided by libfam-atomic that bypass the CPU and the cache.

12.6.9. IMPLEMENTATION STATUS

1. rvma_event_poll(3), rvma_getx(3), and rvma_putx(3) are unimplemented, but will be implemented for June 2016.
2. The TCP implementation requires work to enforce access controls and to implement the new APIs. This should be done by the end of January, 2016.
3. The RDMA implementation has several limitations, but, given the hardware is currently out of plan for June, 2016, we aren’t planning to address any of them before then.

12.6.10. EXAMPLE

The following the rvma.h header file and a simple client-server example that does everything required to transfer data.

12.6.11. rvma.h

```
#ifndef _RVMA_H_
#define _RVMA_H_

#include <stdbool.h>
#include <stdint.h>

#include <sys/types.h>

#include <rvma_errno.h>

#ifdef  __cplusplus
extern "C" {
#endif

/**
 * Type definitions
 *
 * Most are opaque to the user and the actual structures are defined in
 * in the library's internal header, rvma_private.h Some are "counted
 * objects", which have have use counts that allow them to be shared by
 * multiple threads safely. When needed, counted objects have get/put
 * routines in rvma_thread.h to manage the use counts.
 */

struct rvma_root {
    void                        *addr;
    size_t                      len;
    struct rvma_event           *event;
};

struct rvma_event {
    void                        *user;
};

struct rvma_ctxt;

struct rvma_listener;

struct rvma_auth;

struct rvma_access;

#define RVMA_ACCESS_NONE        (0x00)  /* Range cannot be accessed */
#define RVMA_ACCESS_RD          (0x01)  /* Range can be read */
#define RVMA_ACCESS_WR          (0x02)  /* Range can be written */
#define RVMA_ACCESS_LOCAL       (0x04)  /* Accessible by local process only */
#define RVMA_ACCESS_VALID       (0x07)  /* All valid flags */

/* Routines for allocating/destroying types */

extern struct rvma_event *rvma_event_alloc(void);
```

```
extern int rvma_event_destroy(struct rvma_event *event);

extern int rvma_access_add_range(struct rvma_access **paccess,
                                 void *addr, size_t len, int flags);

extern void rvma_access_free(struct rvma_access *access);

extern int rvma_ctxt_access_add_range(struct rvma_ctxt *ctxt,
                                      void *addr, size_t len, int flags);

extern int rvma_ctxt_destroy(struct rvma_ctxt *ctxt);

extern int rvma_listener_destroy(struct rvma_listener *listener);

/* Access routines for opaque types */

extern struct rvma_event *rvma_ctxt_event(struct rvma_ctxt *ctxt);

extern struct rvma_root *rvma_ctxt_remote_root(struct rvma_ctxt *ctxt);

extern int rvma_ctxt_remote_addr_str(const struct rvma_ctxt *ctxt,
                                     char *buf, size_t buf_len);

#define RVMA_ADDRSTRLEN         (80)

extern struct rvma_event *rvma_listener_event(struct rvma_listener *listener);

extern struct rvma_event *rvma_listener_event(struct rvma_listener *listener);

/* Context creation and support functions */

extern int rvma_event_wait(struct rvma_event *event, int nacks);

extern int rvma_event_wait_timeout(struct rvma_event *event, int nacks,
                                   int timeout, int *npend);

extern int rvma_event_wait_abstime(struct rvma_event *event, int nacks,
                                   const struct timespec *abstime, int *npend);

extern int rvma_event_test(struct rvma_event *event, int nacks, int *npend);

extern int rvma_event_waitany_timeout(struct rvma_event **events, int nevents,
                                      int timeout);

extern int rvma_event_waitany_abstime(struct rvma_event **events, int nevents,
                                      const struct timespec *abstime);

extern int rvma_event_waitall_timeout(struct rvma_event **events, int nevents,
                                      int timeout);

extern int rvma_event_waitall_abstime(struct rvma_event **events,
                                      int nevents,
                                      const struct timespec *abstime);

extern int rvma_epoch_new(void);

extern int rvma_epoch_wait_timeout(int timeout);

extern int rvma_epoch_wait_abstime(const struct timespec *abstime);
```

```
extern void rvma_ctxt_create_async(const char *server, const char *service,
                                   const struct rvma_auth *auth,
                                   const struct rvma_access *access,
                                   void *client_root_addr,
                                   size_t client_root_len,
                                   struct rvma_ctxt **ctxt, int *status,
                                   struct rvma_event *done_event);

extern int rvma_ctxt_create(const char *server, const char *service,
                            struct rvma_auth *auth, struct rvma_access *access,
                            void *client_root_addr,
                            size_t client_root_len, struct rvma_ctxt **ctxt);

extern int rvma_ctxt_close(struct rvma_ctxt *ctxt);

extern int rvma_listener_create(const char *service,
                                const struct rvma_auth *auth,
                                int backlog, struct rvma_listener **listener);

extern int rvma_listener_ctxt_get(struct rvma_listener *listener, bool wait,
                                  struct rvma_ctxt **ctxt);

extern int rvma_listener_ctxt_accept(struct rvma_ctxt *ctxt,
                                     const struct rvma_access *access,
                                     void *server_root_addr,
                                     size_t server_root_len);

extern void rvma_event_signal_async(struct rvma_ctxt *ctxt,
                                    struct rvma_event *target_event,
                                    int *status,
                                    struct rvma_event *done_event);

extern int rvma_event_signal(struct rvma_ctxt *ctxt,
                             struct rvma_event *target_event);

extern void rvma_put_async(struct rvma_ctxt *ctxt, void *rdst,
                           const void *lsrc, size_t len,
                           struct rvma_event *source_free,
                           struct rvma_event *rdone_event,
                           int *status, struct rvma_event *done_event);

extern int rvma_put(struct rvma_ctxt *ctxt, void *rdst, const void *lsrc,
                    size_t len, struct rvma_event *source_free,
                    struct rvma_event *rdone_event);

extern void rvma_get_async(void *ldst, struct rvma_ctxt *ctxt,
                           const void *rsrc, size_t len,
                           struct rvma_event *source_free,
                           int *status, struct rvma_event *done_event);

extern int rvma_get(void *ldst, struct rvma_ctxt *ctxt, const void *rsrc,
                    size_t len, struct rvma_event *source_free);

enum rvma_atomic_op {
    RVMA_AOP_MIN,
    RVMA_AOP_ADD = RVMA_AOP_MIN,
    RVMA_AOP_SUB,
    RVMA_AOP_AND,
    RVMA_AOP_OR,
    RVMA_AOP_XOR,
    RVMA_AOP_XCG,
```

```
    RVMA_AOP_CAS,
    RVMA_AOP_MAX = RVMA_AOP_CAS,
};

extern void rvma_atomic_fetch_op_async(struct rvma_ctxt *ctxt,
                                       void *rdst, enum rvma_atomic_op op,
                                       size_t oplen, const void *operand,
                                       void *original, int *status,
                                       struct rvma_event *done_event);

extern int rvma_atomic_fetch_op(struct rvma_ctxt *ctxt, void *rdst,
                                enum rvma_atomic_op op, size_t oplen,
                                const void *operand, void *original);

/* Misc. helpers */

static inline int rvma_error_update(int ret, int new_error)
{
    return (new_error < 0 && ret >= 0 ? new_error : ret);
}

#define rvma_update_error rvma_error_update

static inline void rvma_error_cas(int *ret, int new_error)
{
    int old;
    int cur;

    if (new_error < 0) {
        old = *ret;
        for (;;) {
            if (old < 0)
                break;
            cur = __sync_val_compare_and_swap(ret, old, new_error);
            if (cur == old)
                break;
            old = cur;
        }
    }
}

#define rvma_cas_error rvma_error_cas

#ifdef  __cplusplus
}
#endif

#endif /* _RVMA_H_ */
```

12.6.12. rvma_simple.c

```
#include <errno.h>
#include <libgen.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#include <rvma.h>

static const char *appname;
```

```
/* Use application name for service name (good idea for tests, only) */
#define SVC_NAME    appname
/* We're going to run one server and exit; we don't need more. */
#define SVC_BACKLOG (1)

#define CHECK_ERROR(_status)                                        \
do {                                                                \
    int __status = (_status);                                       \
    if (__status < 0) {                                             \
        fprintf(stderr, "%s:%s,%d:error %d:%s\n",                   \
                appname, __FUNCTION__, __LINE__, __status,          \
                rvma_strerror(__status));                           \
        goto done;                                                  \
    }                                                               \
} while (0)

/* Root data passed by server to client. */
struct svr_root {
    void  *buf;
};

static void usage(void) __attribute__ ((__noreturn__));

static void usage(void)
{
    fprintf(stderr,
        "Usage:%s [hostnameorIP]\n"
        "    Start a server process with no parameters; and then a client\n"
        "    process to connect to it.\n",
        appname);

    exit(255);
}

static int do_server(void)
{
    /* Server, create listener. */
    int ret = 1;                    /* Assume error. */
    /* Init RVMA pointers to NULL in case we have to clean up. */
    struct rvma_listener *listener = NULL;
    struct rvma_access *access = NULL;
    struct rvma_ctxt *ctxt = NULL;
    int buf[2] = { 0, 0 };          /* Data buffer */
    struct svr_root sroot;          /* Root data passed to client. */
    int rc;

    /* Create listener. */
    rc = rvma_listener_create(SVC_NAME, NULL, SVC_BACKLOG, &listener);
    CHECK_ERROR(rc);
    /* Wait for a new context. */
    rc = rvma_listener_ctxt_get(listener, true, &ctxt);
    CHECK_ERROR(rc);
    /* Create access for buffer. (Required for RDMA transports.) */
    rc = rvma_access_add_range(&access, buf, sizeof(buf),
                               RVMA_ACCESS_RD | RVMA_ACCESS_WR);
    CHECK_ERROR(rc);
    sroot.buf = buf;
    /* Accept context, pass access and sroot. */
    rc = rvma_listener_ctxt_accept(ctxt, access, &sroot, sizeof(sroot));
    CHECK_ERROR(rc);
```

```
    /* Context fully created, wait for two signals from client. */
    rc = rvma_event_wait(rvma_ctxt_event(ctxt), 2);
    CHECK_ERROR(rc);
    /* Output the data */
    printf("%s:should be (1, 2): (%d, %d)\n", appname, buf[0], buf[1]);
    /* Do orderly close */
    rc = rvma_ctxt_close(ctxt);
    ctxt = NULL;
    CHECK_ERROR(rc);

    ret = 0;
done:
    /* Close/destroy ctxt first: stops all operations. */
    rvma_ctxt_destroy(ctxt);
    /* access could have been freed immediately after ctxt_accept. */
    rvma_access_free(access);
    /* listener could have been destroyed immediately after ctxt_get. */
    rvma_listener_destroy(listener);

    return ret;
}

static int do_client(const char *host)
{
    /* Client, create context. */
    int ret = 1;                    /* Assume error. */
    /* Init RVMA pointers to NULL in case we have to clean up. */
    struct rvma_access *access = NULL;
    struct rvma_ctxt *ctxt = NULL;
    struct rvma_event *done_event = NULL;
    int buf[2] = { 1, 2 };          /* Data buffer */
    struct rvma_root *rroot;        /* Remote root data from RVMA. */
    struct rvma_event *sevent;      /* Event server will wait on. */
    struct svr_root *sroot;         /* Pointer root data passed by server. */
    int *sbuf;                      /* Typed pointer to server data buffer. */
    int status;                     /* Async operation status. */
    int rc;

    /* Allocate done_event for asynchrounous operations. */
    done_event = rvma_event_alloc();
    rc = (done_event ? 0 : -errno);
    CHECK_ERROR(rc);
    /* Create access for buffer. (Required for RDMA transports.) */
    rc = rvma_access_add_range(&access, buf, sizeof(buf),
                               RVMA_ACCESS_RD | RVMA_ACCESS_WR);
    CHECK_ERROR(rc);
    /* Create context between client and server, synchronous version, */
    rc = rvma_ctxt_create(host, SVC_NAME, NULL, access, NULL, 0, &ctxt);
    CHECK_ERROR(rc);
    /* Context fully created, get pointers to server data. */
    rroot = rvma_ctxt_remote_root(ctxt);
    sevent = rroot->event;
    sroot = rroot->addr;
    sbuf = sroot->buf;
    /*
     * Send buffer to server as two separate asynchronous operations.
     * NOTES:
     * 1.) status must be initialized to zero before first call and
     * should not be altered by the program while the operations are in
     * progress.
     * 2.) sevent on the server will be signaled twice and the server
```

```
     * must wait for both signals because there is no ordering guarantee
     * between operations.
     * 3.) The arguments status and done_event could be NULL and
     * rvma_event_wait() could be replaced with rvma_epoch_wait().
     */
    status = 0;
    rvma_put_async(ctxt, &sbuf[0], &buf[0], sizeof(sbuf[0]), NULL, sevent,
                   &status, done_event);
    rvma_put_async(ctxt, &sbuf[1], &buf[1], sizeof(sbuf[1]), NULL, sevent,
                   &status, done_event);
    /* Wait for both puts to complete. */
    rc = rvma_event_wait(done_event, 2);
    CHECK_ERROR(rc);
    CHECK_ERROR(status);
    rc = rvma_ctxt_close(ctxt);
    ctxt = NULL;
    CHECK_ERROR(rc);

    ret = 0;
done:
    /* Close/destroy ctxt first: stops all operations. */
    rvma_ctxt_destroy(ctxt);
    /* Then free/destroy things that might be using/used by it. */
    rvma_event_destroy(done_event);
    /* access could have been freed immediately after ctxt_create. */
    rvma_access_free(access);

    return ret;
}

int main(int argc, char *argv[])
{
    int ret;

    appname = basename(argv[0]);

    if (argc > 2)
        usage();

    if (argc == 1)
        ret = do_server();
    else
        ret = do_client(argv[1]);

    return ret;
}
```

12.6.13. FILES

`/etc/rvma/lib.conf` allows transports to be enabled or disabled for all processes running on the OS instance. The file is simply an ordered list of the backend libraries which can be reordered or commented out with a "#". The current contents are:

```
    # Transport library priority ordering
    librvmardma.so
    librvmatcp.so
```

12.6.14. SEE ALSO

rvma_perf_bw(1), rvma_perf_lat(1), rvma_atomic_fetch_op(3), rvma_atomic_fetch_opx(3), rvma_atomic_fetch_op_async(3), rvma_atomic_fetch_opx_async(3), rvma_access_add_range(3), rvma_access_free(3), rvma_ctxt_access_add_range(3), rvma_ctxt_close(3), rvma_ctxt_destroy(3), rvma_ctxt_create(3), rvma_ctxt_create_async(3), rvma_ctxt_dec_ref(3), rvma_ctxt_event(3), rvma_ctxt_inc_ref(3), rvma_ctxt_remote_addr_str(3), rvma_ctxt_remote_root(3), rvma_epoch_new(3), rvma_epoch_wait_abstime(3), rvma_epoch_wait_timeout(3), rvma_error_cas(3), rvma_error_update(3) rvma_event_alloc(3), rvma_event_dec_ref(3), rvma_event_destroy(3), rvma_event_inc_ref(3), rvma_event_poll(3), rvma_event_signal(3), rvma_event_signal_async(3), rvma_event_test(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3), rvma_get(3), rvma_getx(3), rvma_get_async(3), rvma_getx_async(3), rvma_listener_create(3), rvma_listener_ctxt_accept(3), rvma_listener_ctxt_get(3), rvma_listener_dec_ref(3), rvma_listener_destroy(3), rvma_listener_event(3), rvma_listener_inc_ref(3), rvma_put(3), rvma_putx(3), rvma_put_async(3), rvma_putx_async(3), rvma_strerror(3),

12.6.15. rvma_access_add_range(3)

NAME

rvma_access_add_range - add region to access descriptor

SYNOPSIS

#include <rvma.h>

int rvma_access_add_range(struct rvma_access ********_access_**, void** ******_addr_**, size_t** _len_**, int** _flags_**);**

Link with **-**_lrvma_

DESCRIPTION

Context creation allows the each of the processes to limit the access the other process has to its memory. Both rvma_ctxt_create_async(3) and rvma_listener_ctxt_accept(3) accept an *access* argument to define these limits.

An access list **struct rvma_access** * is built up though multiple calls to rvma_access_add_range(3) and then passed to rvma_ctxt_create(3) or rvma_listener_ctxt_accept(3). The access list must be stable until the functions complete and the data is copied into the context; once the function returns, changes to the access list will not affect the context and the access list should be freed with rvma_access_free(3). After a context is fully created, new access ranges can be added with rvma_access_add_range(3).

See RVMA(3) [RVMA.3.xml#ACCESS] for an example.

access (input/output): Pointer to access descriptor pointer.

addr (input): Starting virtual address in process of range to add.

len (input): Length of range to add.

flags (input): Flags defining the type of access permitted via RVMA.

- RVMA_ACCESS_NONE : Range cannot be accessed
- RVMA_ACCESS_RD : Range can be read
- RVMA_ACCESS_WR : Range can be written
- RVMA_ACCESS_LOCAL : Range accessible by local process only; a remote process cannot specify these addresses as a source or destination.

RETURN VALUE

Returns the completion status. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_access_free(3), rvma_ctxt_access_add_range(3), rvma_ctxt_create_async(3), rvma_listener_ctxt_accept(3),

12.6.16. rvma_access_free(3)

NAME

rvma_access_free - free access descriptor

SYNOPSIS

#include <rvma.h>

void rvma_access_free(struct rvma_access **access****);**

Link with *-lrvma*

DESCRIPTION

Free an access descriptor created by rvma_access_add_range(3)

access (input): Pointer to access descriptor to free. A NULL pointer is a no-op.

RETURN VALUE

None.

SEE ALSO

RVMA(3), rvma_access_add_range(3)

12.6.17. rvma_atomic_fetch_op_async(3)

NAME

rvma_atomic_fetch_op, rvma_atomic_fetch_opx, rvma_atomic_fetch_op_async, rvma_atomic_fetch_opx_async - perform atomic operations via RVMA

SYNOPSIS

#include <rvma.h>

void rvma_atomic_fetch_op(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***);**

void rvma_atomic_fetch_opx(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***);**

void rvma_atomic_fetch_op_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** **status***, struct rvma_event** **done_event***);**

void rvma_atomic_fetch_opx_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***, int** **status***, struct rvma_event** **done_event***);**

Link with *-lrvma*

DESCRIPTION

Perform an atomic operation in the remote process context via RVMA; this atomic will be done with either CPU instructions and will be incompatible with FAM atomic operations performed with **libfam-atomic**.

rvma_atomic_fetch_op() is a synchronous convenience wrapper around **rvma_atomic_fetch_op_async()**. The "opx" variants have an additional **flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process for op.

op (input): Constant specifying operation type.

- RVMA_AOP_ADD
- RVMA_AOP_SUB

- RVMA_AOP_AND
- RVMA_AOP_OR
- RVMA_AOP_XOR
- RVMA_AOP_XCG : exchange
- RVMA_AOP_CAS : compare and swap

oplen (input): Length of the operands in bytes: 1, 2, 4, 8 (16 for XCG or CAS)

operand (input) : Pointer to a signed/unsigned integer of size *oplen*; the integer contains the value add/subtract/... to the integer pointed to by *rdst*.

original (input/output): Pointer to a signed/unsigned integer of size *oplen*; used to return the original value of the integer pointed to by *rdst*. For compare-and-swap (CAS), the integer is used to provide the expected value of *rdst*. For async variant, *original* will be valid only after *done_event* is signaled.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_atomic_fetch_op_async() returns void.

rvma_atomic_fetch_op() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3)

12.6.18. rvma_atomic_fetch_op_async(3)

NAME

rvma_atomic_fetch_op, rvma_atomic_fetch_opx, rvma_atomic_fetch_op_async, rvma_atomic_fetch_opx_async - perform atomic operations via RVMA

SYNOPSIS

#include <rvma.h>

void rvma_atomic_fetch_op(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***);**

void rvma_atomic_fetch_opx(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***);**

void rvma_atomic_fetch_op_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** **status***, struct rvma_event** **done_event***);**

void rvma_atomic_fetch_opx_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***, int** **status***, struct rvma_event** **done_event***);**

Link with *-lrvma*

DESCRIPTION

Perform an atomic operation in the remote process context via RVMA; this atomic will be done with either CPU instructions and will be incompatible with FAM atomic operations performed with **libfam-atomic**.

rvma_atomic_fetch_op() is a synchronous convenience wrapper around **rvma_atomic_fetch_op_async()**. The "opx" variants have an additional **flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process for op.

op (input): Constant specifying operation type.

- RVMA_AOP_ADD
- RVMA_AOP_SUB

- RVMA_AOP_AND
- RVMA_AOP_OR
- RVMA_AOP_XOR
- RVMA_AOP_XCG : exchange
- RVMA_AOP_CAS : compare and swap

oplen (input): Length of the operands in bytes: 1, 2, 4, 8 (16 for XCG or CAS)

operand (input) : Pointer to a signed/unsigned integer of size *oplen*; the integer contains the value add/subtract/... to the integer pointed to by *rdst*.

original (input/output): Pointer to a signed/unsigned integer of size *oplen*; used to return the original value of the integer pointed to by *rdst*. For compare-and-swap (CAS), the integer is used to provide the expected value of *rdst*. For async variant, *original* will be valid only after *done_event* is signaled.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_atomic_fetch_op_async() returns void.

rvma_atomic_fetch_op() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3)

12.6.19. rvma_atomic_fetch_op_async(3)

NAME

rvma_atomic_fetch_op, rvma_atomic_fetch_opx, rvma_atomic_fetch_op_async, rvma_atomic_fetch_opx_async - perform atomic operations via RVMA

SYNOPSIS

#include <rvma.h>

void rvma_atomic_fetch_op(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***);**

void rvma_atomic_fetch_opx(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***);**

void rvma_atomic_fetch_op_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** **status***, struct rvma_event** **done_event***);**

void rvma_atomic_fetch_opx_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***, int** **status***, struct rvma_event** **done_event***);**

Link with *-lrvma*

DESCRIPTION

Perform an atomic operation in the remote process context via RVMA; this atomic will be done with either CPU instructions and will be incompatible with FAM atomic operations performed with **libfam-atomic**.

rvma_atomic_fetch_op() is a synchronous convenience wrapper around **rvma_atomic_fetch_op_async()**. The "opx" variants have an additional **flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process for op.

op (input): Constant specifying operation type.

- RVMA_AOP_ADD
- RVMA_AOP_SUB

- RVMA_AOP_AND
- RVMA_AOP_OR
- RVMA_AOP_XOR
- RVMA_AOP_XCG : exchange
- RVMA_AOP_CAS : compare and swap

oplen (input): Length of the operands in bytes: 1, 2, 4, 8 (16 for XCG or CAS)

operand (input) : Pointer to a signed/unsigned integer of size *oplen*; the integer contains the value add/subtract/... to the integer pointed to by *rdst*.

original (input/output): Pointer to a signed/unsigned integer of size *oplen*; used to return the original value of the integer pointed to by *rdst*. For compare-and-swap (CAS), the integer is used to provide the expected value of *rdst*. For async variant, *original* will be valid only after *done_event* is signaled.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_atomic_fetch_op_async() returns void.

rvma_atomic_fetch_op() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3)

12.6.20. rvma_atomic_fetch_op_async(3)

NAME

rvma_atomic_fetch_op, rvma_atomic_fetch_opx, rvma_atomic_fetch_op_async, rvma_atomic_fetch_opx_async - perform atomic operations via RVMA

SYNOPSIS

#include <rvma.h>

void rvma_atomic_fetch_op(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***);**

void rvma_atomic_fetch_opx(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***);**

void rvma_atomic_fetch_op_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** **status***, struct rvma_event** **done_event***);**

void rvma_atomic_fetch_opx_async(struct rvma_ctxt **ctxt***, void** *rdst**, enum rvma_atomic_op** *op***, size_t** *oplen***, const void** **operand***, void** **original***, int** *flags***, int** **status***, struct rvma_event** **done_event***);**

Link with *-lrvma*

DESCRIPTION

Perform an atomic operation in the remote process context via RVMA; this atomic will be done with either CPU instructions and will be incompatible with FAM atomic operations performed with **libfam-atomic**.

rvma_atomic_fetch_op() is a synchronous convenience wrapper around **rvma_atomic_fetch_op_async()**. The "opx" variants have an additional **flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process for op.

op (input): Constant specifying operation type.

- RVMA_AOP_ADD
- RVMA_AOP_SUB

- RVMA_AOP_AND
- RVMA_AOP_OR
- RVMA_AOP_XOR
- RVMA_AOP_XCG : exchange
- RVMA_AOP_CAS : compare and swap

oplen (input): Length of the operands in bytes: 1, 2, 4, 8 (16 for XCG or CAS)

operand (input) : Pointer to a signed/unsigned integer of size *oplen*; the integer contains the value add/subtract/... to the integer pointed to by *rdst*.

original (input/output): Pointer to a signed/unsigned integer of size *oplen*; used to return the original value of the integer pointed to by *rdst*. For compare-and-swap (CAS), the integer is used to provide the expected value of *rdst*. For async variant, *original* will be valid only after *done_event* is signaled.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_atomic_fetch_op_async() returns void.

rvma_atomic_fetch_op() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3)

12.6.21. rvma_ctxt_access_add_range(3)

NAME

rvma_ctxt_access_add_range - add access region to existing context

SYNOPSIS

#include <rvma.h>

int rvma_ctxt_access_add_range(struct rvma_ctxt **ctxt***, void** **addr***, size_t** *len***, int** *flags***);**

Link with **-***lrvma*

DESCRIPTION

Add access range to existing context. See rvma_access_add_range(3) for more details.

*ctxt_* (input): Pointer to context.

addr (input): Starting virtual address in process of range to add.

len (input): Length of range to add.

flags (input): Flags defining the type of access permitted via RVMA.

```
RVMA_ACCESS_NONE   : Range cannot be accessed
RVMA_ACCESS_RD     : Range can be read
RVMA_ACCESS_WR     : Range can be written
RVMA_ACCESS_LOCAL  : Range accessible by local process only;
                     a remote process cannot specify these addresses
                     as a source or destination.
```

RETURN VALUE

Returns the completion status. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_access_add_range(3), rvma_access_free(3), rvma_ctxt_create_async(3), rvma_listener_ctxt_accept(3),

12.6.22. rvma_ctxt_close(3)

NAME

rvma_ctxt_close, rvma_ctxt_destroy - shutdown of a context

SYNOPSIS

#include <rvma.h>

int rvma_ctxt_close(struct rvma_ctxt **ctxt***);**

int rvma_ctxt_destroy(struct rvma_ctxt **ctxt***);**

Link with *-lrvma*

DESCRIPTION

rvma_ctxt_close() attempts an orderly shutdown of a context and should be called after all operations on the context are completed. The context will be marked closed; its reference count decremented; and the context will be freed if the reference count is now zero. The event returned by **rvma_ctxt_event()** will be marked destroyed, but will not be freed until the context is freed.

rvma_ctxt_destroy() forces the context to shutdown and all pending operations may be completed with errors. The context will be marked destroyed; its reference count decremented; and the context will be freed if the reference count is now zero. The event returned by **rvma_ctxt_event()** will be marked destroyed, but will not be freed until the context is freed.

See RVMA(3) [RVMA.3.xml#REFCNT] for a brief discussion of multi-threading and reference counts.

ctxt (input): Pointer to context to be closed/destroyed. If NULL, the call is a no-op.

RETURN VALUE

Both functions return a completion status. A negative value indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_ctxt_event(3), rvma_ctxt_create_async(3)

12.6.23. rvma_ctxt_create_async(3)

NAME

rvma_ctxt_create, rvma_ctxt_create_async - create a context between client and server

SYNOPSIS

#include <rvma.h>

int rvma_ctxt_create(const char **server*, **const char** **service*, **const struct rvma_auth** **auth*, **const struct rvma_access** **access*, **void** **client_root_addr*, **size_t** *client_root_len*, **struct rvma_ctxt** ***ctxt*);

void rvma_ctxt_create_async(const char **server*, **const char** **service*, **const struct rvma_auth** **auth*, **const struct rvma_access** **access*, **void** **client_root_addr*, **size_t** *client_root_len*, **struct rvma_ctxt** ***ctxt*, **int** **status*, **struct rvma_event** **done_event*);

Link with *-lrvma*

DESCRIPTION

Create a context between the client and the specified server and service using the best available supported transport.

rvma_ctxt_create() is a synchronous convenience wrapper around **rvma_ctxt_create_async()**.

server (input): A C-string that names the destination server. The default namespace will be the standard network naming via DNS or IP address. In the future, NULL will mean to use a "well-known" service broker or other namespaces might be implemented.

service (input): A C-string that names the destination’s service; this is a name, not a port number.

auth (input) : A placeholder for future authentication information; must be NULL.

access (input) : Memory access restrictions for context; may be NULL. rvma_access_add_range(3) can be used to add ranges after the context is created.

client_root_addr, *client_root_len* (input) : defines a chunk of service-specific data will be copied and sent to the server. Used to bootstrap the communication process.

ctxt (output) : Returns pointer to newly created context; **ctxt* will be NULL if an error occurred; for async variant, the caller must wait for *done_event* to be signaled before this is valid.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_ctxt_create_async() returns void.

rvma_ctxt_create() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

FILES

`/etc/rvma/lib.conf` allows transports to be enabled or disabled for all processes running on the OS instance.

SEE ALSO

RVMA(3), rvma_ctxt_close(3), rvma_ctxt_destroy(3), rvma_ctxt_event(3), rvma_ctxt_remote_root(3), rvma_event_wait(3), rvma_listener_create(3)

12.6.24. rvma_ctxt_create_async(3)

NAME

rvma_ctxt_create, rvma_ctxt_create_async - create a context between client and server

SYNOPSIS

#include <rvma.h>

int rvma_ctxt_create(const char **server*, **const char** **service*, **const struct rvma_auth** **auth*, **const struct rvma_access** **access*, **void** **client_root_addr*, **size_t** *client_root_len*, **struct rvma_ctxt** ***ctxt*);

void rvma_ctxt_create_async(const char **server*, **const char** **service*, **const struct rvma_auth** **auth*, **const struct rvma_access** **access*, **void** **client_root_addr*, **size_t** *client_root_len*, **struct rvma_ctxt** ***ctxt*, **int** **status*, **struct rvma_event** **done_event*);

Link with *-lrvma*

DESCRIPTION

Create a context between the client and the specified server and service using the best available supported transport.

rvma_ctxt_create() is a synchronous convenience wrapper around **rvma_ctxt_create_async()**.

server (input): A C-string that names the destination server. The default namespace will be the standard network naming via DNS or IP address. In the future, NULL will mean to use a "well-known" service broker or other namespaces might be implemented.

service (input): A C-string that names the destination’s service; this is a name, not a port number.

auth (input) : A placeholder for future authentication information; must be NULL.

access (input) : Memory access restrictions for context; may be NULL. rvma_access_add_range(3) can be used to add ranges after the context is created.

client_root_addr, *client_root_len* (input) : defines a chunk of service-specific data will be copied and sent to the server. Used to bootstrap the communication process.

ctxt (output) : Returns pointer to newly created context; **ctxt* will be NULL if an error occurred; for async variant, the caller must wait for *done_event* to be signaled before this is valid.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_ctxt_create_async() returns void.

rvma_ctxt_create() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

FILES

`/etc/rvma/lib.conf` allows transports to be enabled or disabled for all processes running on the OS instance.

SEE ALSO

RVMA(3), rvma_ctxt_close(3), rvma_ctxt_destroy(3), rvma_ctxt_event(3), rvma_ctxt_remote_root(3), rvma_event_wait(3), rvma_listener_create(3)

12.6.25. rvma_ctxt_inc_ref(3)

NAME

rvma_ctxt_inc_ref, rvma_ctxt_dec_ref, rvma_event_inc_ref, rvma_event_dec_ref, rvma_listener_inc_ref, rvma_listener_dec_ref - increment/decrement structure reference-counts

SYNOPSIS

#include <rvma_thread.h>

void rvma_ctxt_inc_ref(struct rvma_ctxt **ctxt***);**

void rvma_ctxt_dec_ref(struct rvma_ctxt **ctxt***);**

void rvma_event_inc_ref(struct rvma_event **event***);**

void rvma_event_dec_ref(struct rvma_event **event***);**

void rvma_listener_inc_ref(struct rvma_listener **listener***);**

void rvma_listener_dec_ref(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

RVMA is inherently multi-threaded and so a reference-count mechanism is used to prevent objects from being freed while still in-use by threads.

rvma_xxx_inc_ref() increments the reference-count; **rvma_xxx_dec_ref()** decrements the reference-count. If the reference-count becomes zero, the structure will be freed.

RETURN VALUE

None.

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3), rvma_event_alloc(3), rvma_listener_create(3)

12.6.26. rvma_ctxt_close(3)

NAME

rvma_ctxt_close, rvma_ctxt_destroy - shutdown of a context

SYNOPSIS

#include <rvma.h>

int rvma_ctxt_close(struct rvma_ctxt **ctxt***);**

int rvma_ctxt_destroy(struct rvma_ctxt **ctxt***);**

Link with *-lrvma*

DESCRIPTION

rvma_ctxt_close() attempts an orderly shutdown of a context and should be called after all operations on the context are completed. The context will be marked closed; its reference count decremented; and the context will be freed if the reference count is now zero. The event returned by **rvma_ctxt_event()** will be marked destroyed, but will not be freed until the context is freed.

rvma_ctxt_destroy() forces the context to shutdown and all pending operations may be completed with errors. The context will be marked destroyed; its reference count decremented; and the context will be freed if the reference count is now zero. The event returned by **rvma_ctxt_event()** will be marked destroyed, but will not be freed until the context is freed.

See RVMA(3) [RVMA.3.xml#REFCNT] for a brief discussion of multi-threading and reference counts.

ctxt (input): Pointer to context to be closed/destroyed. If NULL, the call is a no-op.

RETURN VALUE

Both functions return a completion status. A negative value indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_ctxt_event(3), rvma_ctxt_create_async(3)

12.6.27. rvma_ctxt_event(3)

NAME

rvma_ctxt_event - return pointer to context’s event

SYNOPSIS

#include <rvma.h>

struct rvma_event *rvma_ctxt_event(struct rvma_ctxt **ctxt***);**

Link with *-lrvma*

DESCRIPTION

Contexts are created with a local event for synchronization purposes and its address is part of the **struct rvma_root** sent to remote process when the context is created. This function is a simple wrapper to return the pointer to this event. This event will be destroyed when the context is destroyed or closed. There is no requirement to use this event, nor any restrictions on using other local events for synchronization the context.

ctxt (input): Pointer to context.

RETURN VALUE

Pointer to context’s event.

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3) rvma_event_test(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3), rvma_event_alloc(3), rvma_listener_create(3)

12.6.28. rvma_ctxt_inc_ref(3)

NAME

rvma_ctxt_inc_ref, rvma_ctxt_dec_ref, rvma_event_inc_ref, rvma_event_dec_ref, rvma_listener_inc_ref, rvma_listener_dec_ref - increment/decrement structure reference-counts

SYNOPSIS

#include <rvma_thread.h>

void rvma_ctxt_inc_ref(struct rvma_ctxt **ctxt***);**

void rvma_ctxt_dec_ref(struct rvma_ctxt **ctxt***);**

void rvma_event_inc_ref(struct rvma_event **event***);**

void rvma_event_dec_ref(struct rvma_event **event***);**

void rvma_listener_inc_ref(struct rvma_listener **listener***);**

void rvma_listener_dec_ref(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

RVMA is inherently multi-threaded and so a reference-count mechanism is used to prevent objects from being freed while still in-use by threads.

rvma_xxx_inc_ref() increments the reference-count; **rvma_xxx_dec_ref()** decrements the reference-count. If the reference-count becomes zero, the structure will be freed.

RETURN VALUE

None.

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3), rvma_event_alloc(3), rvma_listener_create(3)

12.6.29. rvma_ctxt_remote_addr_str(3)

NAME

rvma_ctxt_remote_addr_str - return a C-string with the network address of the remote host

SYNOPSIS

#include <rvma.h>

int rvma_ctxt_remote_addr_str(const struct rvma_ctxt **ctxt*, **char** **buf*, **size_t** *buf_len*);

Link with *-lrvma*

DESCRIPTION

Fill in the buffer a C-string with the network address of the remote host.

ctxt (input): Pointer to context.

buf (output): Pointer to buffer to fill.

buf_len (output): Length of buffer; RVMA_ADDRSTRLEN is the minimum suggested length.

RETURN VALUE

Returns the completion status. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -EOVERFLOW will be returned if the buffer is too small to return the string. Other possible errors may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3)

12.6.30. rvma_ctxt_remote_root(3)

NAME

rvma_ctxt_remote_root - return pointer to remote root information

SYNOPSIS

#include <rvma.h>

struct rvma_root *rvma_ctxt_remote_root(struct rvma_ctxt **ctxt***);**

Link with *-lrvma*

DESCRIPTION

Contexts receive a **struct rvma_root** from the remote process when they are created; this function is a simple wrapper to expose a pointer to it. The pointer will remain valid until the context is freed. The **struct rvma_root** provides the service-specific information necessary to bootstrap communication between processes.

ctxt (input): Pointer to context.

RETURN VALUE

Pointer to context's struct rvma_root.

```
struct rvma_root {
    void                *addr;
    size_t              len;
    struct rvma_event   *event;
};
```

addr is a pointer to a blob of data that was sent from the remote process; *len* is the length of the data; *event* is the remote virtual address of the context's event in the other process. (See

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3), rvma_listener_ctxt_accept(3)

12.6.31. rvma_epoch_new(3)

NAME

rvma_epoch_new - Create a new implicit wait epoch

SYNOPSIS

#include <rvma.h>

int rvma_epoch_new(void);

Link with *-lrvma*

DESCRIPTION

Asynchronous operations that specify NULL for both the *status* and *done_event* arguments use wait epochs for completion. (See rvma_epoch_wait_abstime(3), and rvma_epoch_wait_timeout(3) for more details.) An epoch may be created implicitly by the first asynchronous operation that tries to use one or explicitly by this call; **rvma_epoch_new()** allows new epochs to be created to group operations into different epochs. See RVMA(3) [RVMA.3.xml#ASYNC] for an example of their use.

RETURN VALUE

Returns the completion status. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_epoch_wait_abstime(3) rvma_epoch_wait_timeout(3)

12.6.32. rvma_epoch_wait_abstime(3)

NAME

rvma_epoch_wait_abstime, rvma_epoch_wait_timeout - Wait for the oldest epoch to complete

SYNOPSIS

#include <rvma.h>

int rvma_epoch_wait_abstime(const struct timespec **abstime***);**

int rvma_epoch_wait_timeout(int *timeout***);**

Link with *-lrvma*

DESCRIPTION

Asynchronous operations that do not specify *status* and *done_event* are waited for by using epochs which gather all such operation until rvma_epoch_new(3) or one of the **rvma_epoch_wait_xxx()** functions is done. Epochs are waited for in order of creation, the wait function removes the oldest epoch from the list of epochs and waits for all operations associated with it to complete. If new operations requiring an epoch are started, and no epoch exists, a new epoch will be implicitly created.

For **rvma_epoch_wait_abstime()**, *abstime* specifies an absolute time to which to wait for all operations to complete; if the current time is later than *abstime* and there are still pending operations. -ETIMEDOUT will be returned.

For **rvma_epoch_wait_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and there are still pending operations, -ETIMEDOUT will be returned.

See RVMA(3) [RVMA.3.xml#ASYNC] for an example of their use.

RETURN VALUE

Returns a completion status of all operations and the wait, itself. A negative completion status will be the first error seen and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires and the epoch will be returned to the internal epoch list as the oldest available epoch. -RVMA_ERR_NO_EPOCH means there was no epoch to wait upon. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_epoch_new(3)

12.6.33. rvma_epoch_wait_abstime(3)

NAME

rvma_epoch_wait_abstime, rvma_epoch_wait_timeout - Wait for the oldest epoch to complete

SYNOPSIS

#include <rvma.h>

int rvma_epoch_wait_abstime(const struct timespec **abstime***);**

int rvma_epoch_wait_timeout(int *timeout***);**

Link with *-lrvma*

DESCRIPTION

Asynchronous operations that do not specify *status* and *done_event* are waited for by using epochs which gather all such operation until rvma_epoch_new(3) or one of the **rvma_epoch_wait_xxx()** functions is done. Epochs are waited for in order of creation, the wait function removes the oldest epoch from the list of epochs and waits for all operations associated with it to complete. If new operations requiring an epoch are started, and no epoch exists, a new epoch will be implicitly created.

For **rvma_epoch_wait_abstime()**, *abstime* specifies an absolute time to which to wait for all operations to complete; if the current time is later than *abstime* and there are still pending operations. -ETIMEDOUT will be returned.

For **rvma_epoch_wait_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and there are still pending operations, -ETIMEDOUT will be returned.

See RVMA(3) [RVMA.3.xml#ASYNC] for an example of their use.

RETURN VALUE

Returns a completion status of all operations and the wait, itself. A negative completion status will be the first error seen and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires and the epoch will be returned to the internal epoch list as the oldest available epoch. -RVMA_ERR_NO_EPOCH means there was no epoch to wait upon. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_epoch_new(3)

12.6.34. rvma_error_cas(3)

NAME

rvma_error_cas, rvma_error_update - Helper functions to capture first error

SYNOPSIS

#include <rvma.h>

void rvma_error_cas(int **ret*, **int** *new*);

int rvma_error_update(int *ret*, **int** *new*);

DESCRIPTION

These helper functions are used internally to capture the first error that occurs in a series of operations. **rvma_error_cas()** uses an atomic compare-and-swap to guarantee correctness when multi-threading.

For example:

```
ret = 0;
rc = rvma_get(...);
ret = rvma_error_update(ret, rc);
rc = rvma_get(...);
rvma_error_cas(&ret, rc);
/* ret contains the first error that occurs. */
```

RETURN VALUE

rvma_error_cas() returns void, it updates the error in place.

rvma_error_update() returns the updated error.

SEE ALSO

RVMA(3)

12.6.35. rvma_error_cas(3)

NAME

rvma_error_cas, rvma_error_update - Helper functions to capture first error

SYNOPSIS

#include <rvma.h>

void rvma_error_cas(int **ret*, **int** *new*);

int rvma_error_update(int *ret*, **int** *new*);

DESCRIPTION

These helper functions are used internally to capture the first error that occurs in a series of operations. **rvma_error_cas()** uses an atomic compare-and-swap to guarantee correctness when multi-threading.

For example:

```
ret = 0;
rc = rvma_get(...);
ret = rvma_error_update(ret, rc);
rc = rvma_get(...);
rvma_error_cas(&ret, rc);
/* ret contains the first error that occurs. */
```

RETURN VALUE

rvma_error_cas() returns void, it updates the error in place.

rvma_error_update() returns the updated error.

SEE ALSO

RVMA(3)

12.6.36. rvma_event_alloc(3)

NAME

rvma_event_alloc, rvma_event_destroy - Allocate or destroy an event

SYNOPSIS

#include <rvma.h>

struct rvma_event *rvma_event_alloc(void);

int rvma_event_destroy(struct rvma_event **event****);**

Link with *-lrvma*

DESCRIPTION

rvma_event_alloc() allocates a new event structure.

rvma_event_destroy() marks the event as destroyed; wakes all threads sleeping on the event; decrements the reference count; and will free the event, if the reference count is now zero.

event (input): Pointer to event to be destroyed. If NULL, the call is a no-op.

rvma_event_waitany_timeout(3) will indicate that certain events out of a set of events have been signaled; there needs to be a way for these events to quickly refer back to related data structures in the program, so events expose a user-visible field *user* which can be used to store pointers to the related structures; the rest of the event is opaque.

```
struct rvma_event {
    void        *user;
};
```

RETURN VALUE

rvma_event_alloc() returns the pointer to an event or NULL; if NULL is returned, errno will contain the error.

rvma_event_destroy() returns the completion status. A negative value indicate an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function **rvma_strerror()** will convert errors into a human-readable string.

SEE ALSO

RVMA(3)

12.6.37. rvma_ctxt_inc_ref(3)

NAME

rvma_ctxt_inc_ref, rvma_ctxt_dec_ref, rvma_event_inc_ref, rvma_event_dec_ref, rvma_listener_inc_ref, rvma_listener_dec_ref - increment/decrement structure reference-counts

SYNOPSIS

#include <rvma_thread.h>

void rvma_ctxt_inc_ref(struct rvma_ctxt **ctxt***);**

void rvma_ctxt_dec_ref(struct rvma_ctxt **ctxt***);**

void rvma_event_inc_ref(struct rvma_event **event***);**

void rvma_event_dec_ref(struct rvma_event **event***);**

void rvma_listener_inc_ref(struct rvma_listener **listener***);**

void rvma_listener_dec_ref(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

RVMA is inherently multi-threaded and so a reference-count mechanism is used to prevent objects from being freed while still in-use by threads.

rvma_xxx_inc_ref() increments the reference-count; **rvma_xxx_dec_ref()** decrements the reference-count. If the reference-count becomes zero, the structure will be freed.

RETURN VALUE

None.

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3), rvma_event_alloc(3), rvma_listener_create(3)

12.6.38. rvma_event_alloc(3)

NAME

rvma_event_alloc, rvma_event_destroy - Allocate or destroy an event

SYNOPSIS

#include <rvma.h>

struct rvma_event *rvma_event_alloc(void);

int rvma_event_destroy(struct rvma_event **event*);

Link with *-lrvma*

DESCRIPTION

rvma_event_alloc() allocates a new event structure.

rvma_event_destroy() marks the event as destroyed; wakes all threads sleeping on the event; decrements the reference count; and will free the event, if the reference count is now zero.

event (input): Pointer to event to be destroyed. If NULL, the call is a no-op.

rvma_event_waitany_timeout(3) will indicate that certain events out of a set of events have been signaled; there needs to be a way for these events to quickly refer back to related data structures in the program, so events expose a user-visible field *user* which can be used to store pointers to the related structures; the rest of the event is opaque.

```
struct rvma_event {
    void        *user;
};
```

RETURN VALUE

rvma_event_alloc() returns the pointer to an event or NULL; if NULL is returned, errno will contain the error.

rvma_event_destroy() returns the completion status. A negative value indicate an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function **rvma_strerror()** will convert errors into a human-readable string.

SEE ALSO

RVMA(3)

12.6.39. rvma_ctxt_inc_ref(3)

NAME

rvma_ctxt_inc_ref, rvma_ctxt_dec_ref, rvma_event_inc_ref, rvma_event_dec_ref, rvma_listener_inc_ref, rvma_listener_dec_ref - increment/decrement structure reference-counts

SYNOPSIS

#include <rvma_thread.h>

void rvma_ctxt_inc_ref(struct rvma_ctxt **ctxt***);**

void rvma_ctxt_dec_ref(struct rvma_ctxt **ctxt***);**

void rvma_event_inc_ref(struct rvma_event **event***);**

void rvma_event_dec_ref(struct rvma_event **event***);**

void rvma_listener_inc_ref(struct rvma_listener **listener***);**

void rvma_listener_dec_ref(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

RVMA is inherently multi-threaded and so a reference-count mechanism is used to prevent objects from being freed while still in-use by threads.

rvma_xxx_inc_ref() increments the reference-count; **rvma_xxx_dec_ref()** decrements the reference-count. If the reference-count becomes zero, the structure will be freed.

RETURN VALUE

None.

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3), rvma_event_alloc(3), rvma_listener_create(3)

12.6.40. rvma_event_poll(3)

NAME

rvma_event_poll - Poll event and context for completion

SYNOPSIS

#include <rvma.h>

int rvma_event_poll(struct rvma_event **event***, int** *nacks***, int** *microseconds***, int** **npend***, struct rvma_ctxt** **ctxt***);**

Link with **-***lrvma*

DESCRIPTION

Poll *event* --- no sleeping --- until it is signaled *nacks* times; acknowledge the signals, atomically; and return. *ctxt* is needed to achieve the lowest possible latency, but the function can work without it.

event (input) : Pointer to event to wait upon; may not be NULL.

nacks (input) : The number of signals to wait for; if 0, the wait will return immediately. If nacks is greater than 1, the wait is for all or none; specifically, if the variants with timeouts expire unsatisfied, they will consume no signals on the events.

microseconds (input) : The number of microseconds to poll.

npend (output): Returns the number of pending signals on *event*.

ctxt (input) : The _ctxt processing the operations that will generate the signals, needed for minimal latency, but may be NULL.

RETURN VALUE

Returns the completion status. On success, this will be *nacks*. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if *microseconds* expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_test(3) rvma_event_wait_abstime(3), rvma_event_wait_timeout(3)

12.6.41. rvma_event_signal_async(3)

NAME

rvma_event_signal, rvma_event_signal_async - signal an event

SYNOPSIS

#include <rvma.h>

void rvma_event_signal(struct rvma_ctxt **ctxt***, struct rvma_event** **target_event***);**

void rvma_event_signal_async(struct rvma_ctxt **ctxt***, struct rvma_event** **target_event***, int** **status***, struct rvma_event** **done_event***);**

Link with *-lrvma*

DESCRIPTION

A signal is sent to the event designated by *target_event* and it's signal count is incremented. If a thread is waiting on the event with one of the **rvma_waitxxx()** routines and the signal satisfies the wait condition, the waiting thread is woken up.

For events in the remote process, the remote process must have sent the address of *target_event* to the local process in some manner. For example, the event in the **struct rvma_root** returned by rvma_ctxt_remote_root(3) is a remote event provided for convenience to the programmer.

rvma_event_signal() is a synchronous convenience wrapper around **rvma_event_signal_async()**.

ctxt (input) : Pointer to context; NULL means operation is in local process.

target_event (input): The virtual address in the destination process for the event.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_event_signal_async() returns void.

rvma_event_signal() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_alloc(3), rvma_event_destroy(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3)

12.6.42. rvma_event_signal_async(3)

NAME

rvma_event_signal, rvma_event_signal_async - signal an event

SYNOPSIS

#include <rvma.h>

void rvma_event_signal(struct rvma_ctxt **ctxt***, struct rvma_event** **target_event***);**

void rvma_event_signal_async(struct rvma_ctxt **ctxt***, struct rvma_event** **target_event***, int** **status***, struct rvma_event** **done_event***);**

Link with *-lrvma*

DESCRIPTION

A signal is sent to the event designated by *target_event* and it's signal count is incremented. If a thread is waiting on the event with one of the **rvma_waitxxx()** routines and the signal satisfies the wait condition, the waiting thread is woken up.

For events in the remote process, the remote process must have sent the address of *target_event* to the local process in some manner. For example, the event in the **struct rvma_root** returned by rvma_ctxt_remote_root(3) is a remote event provided for convenience to the programmer.

rvma_event_signal() is a synchronous convenience wrapper around **rvma_event_signal_async()**.

ctxt (input) : Pointer to context; NULL means operation is in local process.

target_event (input): The virtual address in the destination process for the event.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_event_signal_async() returns void.

rvma_event_signal() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_alloc(3), rvma_event_destroy(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3)

12.6.43. rvma_event_test(3)

NAME

rvma_event_test - Test if an event has been signaled and return immediately

SYNOPSIS

#include <rvma.h>

int rvma_event_test(struct rvma_event **event***, int** *nacks***, int** **npend***);**

Link with *-lrvma*

DESCRIPTION

This is a wrapper for rvma_event_wait_timeout(3) with a timeout of 0. Tests *event* to see if at least *nacks* signals are available; if there are, the signals are acknowledged and *nacks* will be returned; if there are not, then -ETIMEDOUT will be returned.

event (input) : Pointer to event to wait upon; may not be NULL.

nacks (input) : The number of signals to test for.

npend (output): Returns the number of unacknowledged signals remaining on *event* when the call exits.

RETURN VALUE

Returns the completion status. On success, this will be *nacks*. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if *nacks* signals are not available. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_poll(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3),

12.6.44. rvma_event_wait(3)

NAME

rvma_event_wait, rvma_event_wait_abstime, rvma_event_wait_timeout - Wait for a single event to be signaled

SYNOPSIS

#include <rvma.h>

int rvma_event_wait(struct rvma_event **event***, int** *nacks***);**

int rvma_event_wait_timeout(struct rvma_event **event***, int** *nacks***, int** *timeout***, int** **npend***);**

int rvma_event_wait_abstime(struct rvma_event **event***, int** *nacks***, const struct timespec** **abstime***, int** **npend***);**

Link with *-lrvma*

DESCRIPTION

Wait for *event* to be signaled *nacks* times; acknowledge the signals; and return.

> Note
>
> Trying to wait on the same event simultaneously on multiple threads has undefined wait behavior and may cause errors to be returned.

event (input) : Pointer to event to wait upon; may not be NULL.

nacks (input) : The number of signals to wait for; if zero, the wait will return immediately. If nacks is greater than 1, the wait is for all or none; specifically, if the variants with timeouts expire unsatisfied, they will consume no signals on the events.

For **rvma_event_wait_abstime()**, *abstime* specifies an absolute time to which to wait for all signals to be received; if the current time is later than *abstime* and there are less than *nacks* signals, -ETIMEDOUT will be returned.

For **rvma_event_wait_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and there are less than *nacks* signals, -ETIMEDOUT will be returned.

npend (output): Returns the number of unacknowledged signals remaining on *event* when the call exits.

RETURN VALUE

Returns the completion status. On success, this will be *nacks*. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_test(3), rvma_event_test(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3)

12.6.45. rvma_event_wait(3)

NAME

rvma_event_wait, rvma_event_wait_abstime, rvma_event_wait_timeout - Wait for a single event to be signaled

SYNOPSIS

#include <rvma.h>

int rvma_event_wait(struct rvma_event **event***, int** *nacks***);**

int rvma_event_wait_timeout(struct rvma_event **event***, int** *nacks***, int** *timeout***, int** **npend***);**

int rvma_event_wait_abstime(struct rvma_event **event***, int** *nacks***, const struct timespec** **abstime***, int** **npend***);**

Link with *-lrvma*

DESCRIPTION

Wait for *event* to be signaled *nacks* times; acknowledge the signals; and return.

> Note
>
> Trying to wait on the same event simultaneously on multiple threads has undefined wait behavior and may cause errors to be returned.

event (input) : Pointer to event to wait upon; may not be NULL.

nacks (input) : The number of signals to wait for; if zero, the wait will return immediately. If nacks is greater than 1, the wait is for all or none; specifically, if the variants with timeouts expire unsatisfied, they will consume no signals on the events.

For **rvma_event_wait_abstime()**, *abstime* specifies an absolute time to which to wait for all signals to be received; if the current time is later than *abstime* and there are less than *nacks* signals, -ETIMEDOUT will be returned.

For **rvma_event_wait_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and there are less than *nacks* signals, -ETIMEDOUT will be returned.

npend (output): Returns the number of unacknowledged signals remaining on *event* when the call exits.

RETURN VALUE

Returns the completion status. On success, this will be *nacks*. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_test(3), rvma_event_test(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3)

12.6.46. rvma_event_waitall_abstime(3)

NAME

rvma_event_waitall_abstime, rvma_event_waitall_timeout - Wait for all events in a given set be signaled

SYNOPSIS

#include <rvma.h>

int rvma_event_waitall_abstime(struct rvma_event ***events*, int** *nevents*, **const struct timespec** ***abstime*);**

int rvma_event_waitall_timeout(struct rvma_event ***events*, int** *nevents*, **int** *timeout*);**

Link with *-lrvma*

DESCRIPTION

Wait for all events in the specified set to be signaled once; acknowledge a single signal on all events; and return.

NOTE: Trying to wait on the same event simultaneously on multiple threads has undefined wait behavior and may cause errors to be returned.

events (input) : Pointer to an array event pointers to wait upon; may not be NULL.

nevents (input) : The number of events specified; if 0 is specified, the call will be a no-op.

For **rvma_event_waitall_abstime()**, *abstime* specifies an absolute time to which to wait for all events to be signal; if the current time is later than *abstime* and some events are still not signaled, -ETIMEDOUT will be returned.

For **rvma_epoch_waitall_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and some events are still not signaled, -ETIMEDOUT will be returned.

RETURN VALUE

Returns the completion status. On success, the number of events acknowledged will be returned. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_poll(3), rvma_event_test(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3)

12.6.47. rvma_event_waitall_abstime(3)

NAME

rvma_event_waitall_abstime, rvma_event_waitall_timeout - Wait for all events in a given set be signaled

SYNOPSIS

#include <rvma.h>

int rvma_event_waitall_abstime(struct rvma_event ***events*, int** *nevents*, **const struct timespec** ***abstime*);**

int rvma_event_waitall_timeout(struct rvma_event ***events*, int** *nevents*, **int** *timeout*);**

Link with *-lrvma*

DESCRIPTION

Wait for all events in the specified set to be signaled once; acknowledge a single signal on all events; and return.

NOTE: Trying to wait on the same event simultaneously on multiple threads has undefined wait behavior and may cause errors to be returned.

events (input) : Pointer to an array event pointers to wait upon; may not be NULL.

nevents (input) : The number of events specified; if 0 is specified, the call will be a no-op.

For **rvma_event_waitall_abstime()**, *abstime* specifies an absolute time to which to wait for all events to be signal; if the current time is later than *abstime* and some events are still not signaled, -ETIMEDOUT will be returned.

For **rvma_epoch_waitall_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and some events are still not signaled, -ETIMEDOUT will be returned.

RETURN VALUE

Returns the completion status. On success, the number of events acknowledged will be returned. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_poll(3), rvma_event_test(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3)

12.6.48. rvma_event_waitany_abstime(3)

NAME

rvma_event_waitany_abstime, rvma_event_waitany_timeout - Wait for some events in a given set be signaled

SYNOPSIS

#include <rvma.h>

int rvma_event_waitany_abstime(struct rvma_event ********events**, int** *nevents*, **const struct timespec** *******abstime***);**

int rvma_event_waitany_timeout(struct rvma_event ********events**, int** *nevents*, **int** *timeout***);**

Link with *-lrvma*

DESCRIPTION

Wait for some events in the specified set to be signaled once; acknowledge a single signal on the signaled events; and return.

NOTE: Trying to wait on the same event simultaneously on multiple threads has undefined wait behavior and may cause errors to be returned.

events (input/output) : Pointer to an array of event pointers to wait upon; may not be NULL. On a successful return, the acknowledged events --- there may be more than one because multiple events may be signaled before the thread is woken by the scheduler --- will be moved to the front of the array.

nevents (input) : The number of events specified; if 0 is specified, the call will be a no-op.

For **rvma_event_waitany_abstime()**, *abstime* specifies an absolute time to which to wait for all events to be signal; if the current time is later than *abstime* and no events have been signaled, -ETIMEDOUT will be returned.

For **rvma_epoch_waitany_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and no events have been signaled, -ETIMEDOUT will be returned.

RETURN VALUE

Returns the completion status. On success, the number of events acknowledged will be returned and the acknowledged events will be moved to the beginning of

the array. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_poll(3), rvma_event_test(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3),

12.6.49. rvma_event_waitany_abstime(3)

NAME

rvma_event_waitany_abstime, rvma_event_waitany_timeout - Wait for some events in a given set be signaled

SYNOPSIS

#include <rvma.h>

int rvma_event_waitany_abstime(struct rvma_event ********events**, int** *nevents*, **const struct timespec** *******abstime***);**

int rvma_event_waitany_timeout(struct rvma_event ********events**, int** *nevents*, **int** *timeout***);**

Link with *-lrvma*

DESCRIPTION

Wait for some events in the specified set to be signaled once; acknowledge a single signal on the signaled events; and return.

NOTE: Trying to wait on the same event simultaneously on multiple threads has undefined wait behavior and may cause errors to be returned.

events (input/output) : Pointer to an array of event pointers to wait upon; may not be NULL. On a successful return, the acknowledged events --- there may be more than one because multiple events may be signaled before the thread is woken by the scheduler --- will be moved to the front of the array.

nevents (input) : The number of events specified; if 0 is specified, the call will be a no-op.

For **rvma_event_waitany_abstime()**, *abstime* specifies an absolute time to which to wait for all events to be signal; if the current time is later than *abstime* and no events have been signaled, -ETIMEDOUT will be returned.

For **rvma_epoch_waitany_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and no events have been signaled, -ETIMEDOUT will be returned.

RETURN VALUE

Returns the completion status. On success, the number of events acknowledged will be returned and the acknowledged events will be moved to the beginning of

the array. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_poll(3), rvma_event_test(3), rvma_event_wait(3), rvma_event_wait_abstime(3), rvma_event_wait_timeout(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3),

12.6.50. rvma_event_wait(3)

NAME

rvma_event_wait, rvma_event_wait_abstime, rvma_event_wait_timeout - Wait for a single event to be signaled

SYNOPSIS

#include <rvma.h>

int rvma_event_wait(struct rvma_event **event***, int** *nacks***);**

int rvma_event_wait_timeout(struct rvma_event **event***, int** *nacks***, int** *timeout***, int** **npend***);**

int rvma_event_wait_abstime(struct rvma_event **event***, int** *nacks***, const struct timespec** **abstime***, int** **npend***);**

Link with *-lrvma*

DESCRIPTION

Wait for *event* to be signaled *nacks* times; acknowledge the signals; and return.

> Note
>
> Trying to wait on the same event simultaneously on multiple threads has undefined wait behavior and may cause errors to be returned.

event (input) : Pointer to event to wait upon; may not be NULL.

nacks (input) : The number of signals to wait for; if zero, the wait will return immediately. If nacks is greater than 1, the wait is for all or none; specifically, if the variants with timeouts expire unsatisfied, they will consume no signals on the events.

For **rvma_event_wait_abstime()**, *abstime* specifies an absolute time to which to wait for all signals to be received; if the current time is later than *abstime* and there are less than *nacks* signals, -ETIMEDOUT will be returned.

For **rvma_event_wait_timeout()**, *timeout* specifies a timeout in milliseconds, with -1 meaning an infinite wait. If the timeout expires, or is 0, and there are less than *nacks* signals, -ETIMEDOUT will be returned.

npend (output): Returns the number of unacknowledged signals remaining on *event* when the call exits.

RETURN VALUE

Returns the completion status. On success, this will be *nacks*. A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. -ETIMEDOUT will be returned if the timeout expires. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_test(3), rvma_event_test(3), rvma_event_waitall_abstime(3), rvma_event_waitall_timeout(3), rvma_event_waitany_abstime(3), rvma_event_waitany_timeout(3)

12.6.51. rvma_get_async(3)

NAME

rvma_get, rvma_getx, rvma_get_async, rvma_getx_async - copy data from the remote process to the local process

SYNOPSIS

#include <rvma.h>

int rvma_get(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*);

int rvma_getx(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, int ***flags***);**

void rvma_get_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** **status*, **struct rvma_event** **done_event*);

void rvma_getx_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

Link with *-lrvma*

DESCRIPTION

Copy data from the remote process associated with the context to the local process.

rvma_get() is a synchronous convenience wrapper around **rvma_get_async()**. The "getx" variants have an additional *flags* argument.

ldst (input): Destination virtual address in the local process.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rsrc (input): Source virtual address in the remote process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the remote process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. With some transports, this may allow better overlap of processing and data transfer.

flags (input, getx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_get_async() returns void.

rvma_get() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.52. rvma_get_async(3)

NAME

rvma_get, rvma_getx, rvma_get_async, rvma_getx_async - copy data from the remote process to the local process

SYNOPSIS

#include <rvma.h>

int rvma_get(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*);

int rvma_getx(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, int ***flags***);**

void rvma_get_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** **status*, **struct rvma_event** **done_event*);

void rvma_getx_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

Link with *-lrvma*

DESCRIPTION

Copy data from the remote process associated with the context to the local process.

rvma_get() is a synchronous convenience wrapper around **rvma_get_async()**. The "getx" variants have an additional *flags* argument.

ldst (input): Destination virtual address in the local process.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rsrc (input): Source virtual address in the remote process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the remote process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. With some transports, this may allow better overlap of processing and data transfer.

flags (input, getx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_get_async() returns void.

rvma_get() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.53. rvma_get_async(3)

NAME

rvma_get, rvma_getx, rvma_get_async, rvma_getx_async - copy data from the remote process to the local process

SYNOPSIS

#include <rvma.h>

int rvma_get(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*);

int rvma_getx(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, int ***flags***);**

void rvma_get_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** **status*, **struct rvma_event** **done_event*);

void rvma_getx_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

Link with *-lrvma*

DESCRIPTION

Copy data from the remote process associated with the context to the local process.

rvma_get() is a synchronous convenience wrapper around **rvma_get_async()**. The "getx" variants have an additional *flags* argument.

ldst (input): Destination virtual address in the local process.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rsrc (input): Source virtual address in the remote process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the remote process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. With some transports, this may allow better overlap of processing and data transfer.

flags (input, getx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_get_async() returns void.

rvma_get() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.54. rvma_get_async(3)

NAME

rvma_get, rvma_getx, rvma_get_async, rvma_getx_async - copy data from the remote process to the local process

SYNOPSIS

#include <rvma.h>

int rvma_get(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*);

int rvma_getx(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, int ***flags***);**

void rvma_get_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** **status*, **struct rvma_event** **done_event*);

void rvma_getx_async(void **ldst*, **struct rvma_ctxt** **ctxt*, **const void** **rsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

Link with *-lrvma*

DESCRIPTION

Copy data from the remote process associated with the context to the local process.

rvma_get() is a synchronous convenience wrapper around **rvma_get_async()**. The "getx" variants have an additional *flags* argument.

ldst (input): Destination virtual address in the local process.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rsrc (input): Source virtual address in the remote process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the remote process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. With some transports, this may allow better overlap of processing and data transfer.

flags (input, getx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_get_async() returns void.

rvma_get() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.55. rvma_listener_create(3)

NAME

rvma_listener_create - create a listener for service requests

SYNOPSIS

#include <rvma.h>

int rvma_listener_create(const char **service***, const struct rvma_auth** **auth*, int ***backlog***, struct rvma_listener ****listener***);**

Link with ***-lrvma***

DESCRIPTION

Create a listener for a specified service on a given host; there may be only one listener for a given service on a host. rvma_listener_ctxt_get(3) is used to get requests for new contexts as they occur. Listeners are destroyed by rvma_listener_destroy(3).

service (input): A C-string that names the service; this is a name, not a port number.

auth (input) : A placeholder for future authentication information; must be NULL.

listener (output) : Returns pointer to newly created listener; **listener* will be NULL if an error occurred.

RETURN VALUE

Returns the completion status; a negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_listener_ctxt_get(3)

12.6.56. rvma_listener_ctxt_accept(3)

NAME

rvma_listener_ctxt_accept - accept a pending context

SYNOPSIS

#include <rvma.h>

void rvma_listener_ctxt_accept(struct rvma_ctxt **ctxt*, **const struct rvma_access** **access*, **void** **server_root_addr*, **size_t** *server_root_len*);

Link with *-lrvma*

DESCRIPTION

Fully accept a pending context returned by rvma_listener_ctxt_get(3)

ctxt (input) : Pointer to context.

access (input) : Memory access restrictions for context; may be NULL. rvma_access_add_range(3) can be used to add ranges after the context is created.

server_root_addr, *server_root_len* (input) : defines a chunk of service-specific data that will be copied and sent to the client. Used to bootstrap the communication process.

RETURN VALUE

Returns the completion status; a negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_ctxt_destroy(3), rvma_listener_ctxt_get(3)

12.6.57. rvma_listener_ctxt_get(3)

NAME

rvma_listener_ctxt_get - get a pending context from a listener

SYNOPSIS

#include <rvma.h>

int rvma_listener_ctxt_get(struct rvma_listener **listener***, bool** *wait***, struct rvma_ctxt** ***ctxt***);**

Link with **-***lrvma*

DESCRIPTION

Get a pending context from listener. The context is incomplete and needs to be either accepted with rvma_listener_ctxt_accept(3) or destroyed with rvma_ctxt_destroy(3),

listener (input) : Pointer to listener.

wait (input) : If true, wait until a context is available. If false, no wait is done and the next available context will be available.

ctxt (output) : Returns pointer to newly created context; **ctxt* will be NULL if no context is available or an error occurred.

RETURN VALUE

Returns the completion status; a negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_ctxt_destroy(3), rvma_listener_ctxt_accept(3)

12.6.58. rvma_ctxt_inc_ref(3)

NAME

rvma_ctxt_inc_ref, rvma_ctxt_dec_ref, rvma_event_inc_ref, rvma_event_dec_ref, rvma_listener_inc_ref, rvma_listener_dec_ref - increment/decrement structure reference-counts

SYNOPSIS

#include <rvma_thread.h>

void rvma_ctxt_inc_ref(struct rvma_ctxt **ctxt***);**

void rvma_ctxt_dec_ref(struct rvma_ctxt **ctxt***);**

void rvma_event_inc_ref(struct rvma_event **event***);**

void rvma_event_dec_ref(struct rvma_event **event***);**

void rvma_listener_inc_ref(struct rvma_listener **listener***);**

void rvma_listener_dec_ref(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

RVMA is inherently multi-threaded and so a reference-count mechanism is used to prevent objects from being freed while still in-use by threads.

rvma_xxx_inc_ref() increments the reference-count; **rvma_xxx_dec_ref()** decrements the reference-count. If the reference-count becomes zero, the structure will be freed.

RETURN VALUE

None.

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3), rvma_event_alloc(3), rvma_listener_create(3)

12.6.59. rvma_listener_destroy(3)

NAME

rvma_listener_destroy - shutdown of a listener

SYNOPSIS

#include <rvma.h>

int rvma_listener_destroy(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

Mark the listener structure as destroyed; destroy any pending connections still on the listener --- any contexts that have been removed from the listener with rvma_listener_ctxt_get(3) are not affected; decrements its reference count; and will free the listener if the reference count is now zero.

listener (input): Pointer to listener to be destroyed. If NULL, the call is a no-op.

RETURN VALUE

Returns a completion status; a negative value indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_listener_create(3),

12.6.60. rvma_listener_event(3)

NAME

rvma_listener_event - return pointer to listener’s event

SYNOPSIS

#include <rvma.h>

struct rvma_event *rvma_listener_event(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

rvma_listener_ctxt_get(3) can wait for until a context request is made, but a process may wish to wait for multiple events instead of just waiting for new requests. This function exposes the internal wait event for use with the various event-wait APIs, such as rvma_event_waitany_timeout(3). The event will be destroyed when the listener is destroyed.

listener (input): Pointer to listener.

RETURN VALUE

Pointer to listener’s event.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_listener_ctxt_get(3)

12.6.61. rvma_ctxt_inc_ref(3)

NAME

rvma_ctxt_inc_ref, rvma_ctxt_dec_ref, rvma_event_inc_ref, rvma_event_dec_ref, rvma_listener_inc_ref, rvma_listener_dec_ref - increment/decrement structure reference-counts

SYNOPSIS

#include <rvma_thread.h>

void rvma_ctxt_inc_ref(struct rvma_ctxt **ctxt***);**

void rvma_ctxt_dec_ref(struct rvma_ctxt **ctxt***);**

void rvma_event_inc_ref(struct rvma_event **event***);**

void rvma_event_dec_ref(struct rvma_event **event***);**

void rvma_listener_inc_ref(struct rvma_listener **listener***);**

void rvma_listener_dec_ref(struct rvma_listener **listener***);**

Link with *-lrvma*

DESCRIPTION

RVMA is inherently multi-threaded and so a reference-count mechanism is used to prevent objects from being freed while still in-use by threads.

rvma_xxx_inc_ref() increments the reference-count; **rvma_xxx_dec_ref()** decrements the reference-count. If the reference-count becomes zero, the structure will be freed.

RETURN VALUE

None.

SEE ALSO

RVMA(3), rvma_ctxt_create_async(3), rvma_event_alloc(3), rvma_listener_create(3)

12.6.62. rvma_perf_bw(1)

NAME

rvma_perf_bw - Test client-server bandwidth

SYNOPSIS

client:

rvma_perf_bw <*hostname_or_IP*> <*message_size*> <*nops*> <put|get|bidir> *[threads]*

server:

rvma_perf_bw

DESCRIPTION

Start a server process with no arguments and then connect the client process to it to report bandwidth in gigabits-per-second.

hostname_or_IP : Network hostname or IP address NOTE: For the initial release of the RDMA backend, the hostname or IP must be bound to a NIC that supports RDMA, this will be fixed in version 2.x.x.

message_size : The size, in bytes, of the messages sent. A decimal number followed by an optional character from "kmgKMG" specifying a multiplier, where lower-case letters represent decimal kilobytes, megabytes, and gigabytes and upper-case letters represent kibibytes, mebibytes, and gibibytes.

nops : Number of messages sent; may have a multiplier, as in *message_size*.

put|get|bidir : Direction of operations.

threads : Number of threads performing operations: one, if unspecified.

Note

The amount of data sent/received is *message_size* * *nops* * *threads*. Using *bidir* doubles this number.

SEE ALSO

RVMA(3), rvma_perf_lat(1)

12.6.63. rvma_perf_lat(1)

NAME

rvma_perf_lat - Test client-server latency

SYNOPSIS

client:

rvma_perf_lat <*hostname_or_IP*> <*message_size*> <*nops*> <put|get|bidir> *[threads]*

server:

rvma_perf_lat

DESCRIPTION

Start a server process with no arguments and then connect the client process to it to report the minimum, maximum, and average latency in microseconds.

hostname_or_IP : Network hostname or IP address NOTE: For the initial release of the RDMA backend, the hostname or IP must be bound to a NIC that supports RDMA, this will be fixed in version 2.x.x.

message_size : The size, in bytes, of the messages sent. A decimal number followed by an optional character from "kmgKMG" specifying a multiplier, where lower-case letters represent decimal kilobytes, megabytes, and gigabytes and upper-case letters represent kibibytes, mebibytes, and gibibytes.

nops : Number of messages sent; may have a multiplier, as in *message_size*.

put|get|bidir : Direction of operations.

threads : Number of threads performing operations: one, if unspecified.

Note

The amount of data sent/received is *message_size* * *nops* * *threads*. Using *bidir* doubles this number.

SEE ALSO

RVMA(3), rvma_perf_bw(1)

12.6.64. rvma_put_async(3)

NAME

rvma_put_async, rvma_put, rvma_putx_async, rvma_putx - copy data from the local process to the remote process

SYNOPSIS

#include <rvma.h>

void rvma_put_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_put(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*);

void rvma_putx_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_putx(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*);

Link with *-lrvma*

DESCRIPTION

Copy data from the local process to the remote process associated with the context.

rvma_put() is a synchronous convenience wrapper around **rvma_put_async()**. The "putx" variants have an additional *flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process.

lsrc (input): Source virtual address in the local process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the local process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. This may allow better overlap of processing and data transfer.

rdone_event (input) : The virtual address of an event in the remote process that will be signaled once the data is successfully delivered. With some transports, combining the signal delivery with the data transfer may be possible and will reduce overhead.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_put_async() returns void.

rvma_put() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.65. rvma_put_async(3)

NAME

rvma_put_async, rvma_put, rvma_putx_async, rvma_putx - copy data from the local process to the remote process

SYNOPSIS

#include <rvma.h>

void rvma_put_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_put(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*);

void rvma_putx_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_putx(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*);

Link with *-lrvma*

DESCRIPTION

Copy data from the local process to the remote process associated with the context.

rvma_put() is a synchronous convenience wrapper around **rvma_put_async()**. The "putx" variants have an additional *flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process.

lsrc (input): Source virtual address in the local process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the local process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. This may allow better overlap of processing and data transfer.

rdone_event (input) : The virtual address of an event in the remote process that will be signaled once the data is successfully delivered. With some transports, combining the signal delivery with the data transfer may be possible and will reduce overhead.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_put_async() returns void.

rvma_put() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.66. rvma_put_async(3)

NAME

rvma_put_async, rvma_put, rvma_putx_async, rvma_putx - copy data from the local process to the remote process

SYNOPSIS

#include <rvma.h>

void rvma_put_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_put(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*);

void rvma_putx_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_putx(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*);

Link with *-lrvma*

DESCRIPTION

Copy data from the local process to the remote process associated with the context.

rvma_put() is a synchronous convenience wrapper around **rvma_put_async()**. The "putx" variants have an additional *flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process.

lsrc (input): Source virtual address in the local process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the local process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. This may allow better overlap of processing and data transfer.

rdone_event (input) : The virtual address of an event in the remote process that will be signaled once the data is successfully delivered. With some transports, combining the signal delivery with the data transfer may be possible and will reduce overhead.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_put_async() returns void.

rvma_put() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.67. rvma_put_async(3)

NAME

rvma_put_async, rvma_put, rvma_putx_async, rvma_putx - copy data from the local process to the remote process

SYNOPSIS

#include <rvma.h>

void rvma_put_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_put(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*);

void rvma_putx_async(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*, **int** **status*, **struct rvma_event** **done_event*);

int rvma_putx(struct rvma_ctxt **ctxt*, **void** **rdst*, **const void** **lsrc*, **size_t** *len*, **struct rvma_event** **source_free*, **struct rvma_event** **rdone_event*, **int** *flags*);

Link with *-lrvma*

DESCRIPTION

Copy data from the local process to the remote process associated with the context.

rvma_put() is a synchronous convenience wrapper around **rvma_put_async()**. The "putx" variants have an additional *flags* argument.

ctxt (input) : Pointer to context; NULL means operation is in local process.

rdst (input): Destination virtual address in the remote process.

lsrc (input): Source virtual address in the local process.

len (input): Length of data to be transferred; if zero, no data is transferred, but any signals are sent.

source_free (input) : The virtual address of an event in the local process that will be signaled once the data is in flight and the source buffer will no longer be accessed; may be NULL. This may allow better overlap of processing and data transfer.

rdone_event (input) : The virtual address of an event in the remote process that will be signaled once the data is successfully delivered. With some transports, combining the signal delivery with the data transfer may be possible and will reduce overhead.

flags (input, putx only) : Controls extended features; the only valid values are zero or one of the values defined below:

- RVMA_XFLAGS_PERSIST_DST : Persist destination memory.

status (output, async only) : Returns completion status; not valid until *done_event* is signaled. (See RVMA(3) [RVMA.3.xml#ASYNC] for a complete description of how *status* and *done_event* are used correctly.)

done_event (input, async only) : Pointer to event signaled in local process when operation complete.

RETURN VALUE

rvma_put_async() returns void.

rvma_put() returns the completion status.

A negative completion status indicates an error and will either be the a negative standard *errno* or a RVMA specific error value. The exact errors that are possible may depend on the underlying transport. The function rvma_strerror(3) will convert errors into a human-readable string.

SEE ALSO

RVMA(3), rvma_event_wait(3), rvma_put_async(3)

12.6.68. rvma_strerror(3)

NAME

rvma_strerror - Return a human-readable string for RVMA errors

SYNOPSIS

#include <rvma.h>

const char *rvma_strerror(int *status*);

Link with *-lrvma*

DESCRIPTION

Returns a human readable string for RVMA error status. This is either a string returned by the standard **strerror()** function or strings for RVMA specific errors. No internationalization is done for RVMA specific errors at this time.

status (input) : Completion status. If greater than or equal to 0, the string "success" is returned; if negative, then the appropriate error string is returned. The following RVMA specific errors are currently defined in **rvma_errno.h**:

- RVMA_ERR_OBJ_INVALID : RVMA object is invalid
- RVMA_ERR_OBJ_DESTROYED : RVMA object is destroyed
- RVMA_ERR_NO_FABRIC : No RVMA fabric found
- RVMA_ERR_EVT_OVERFLOW : RVMA event overflow
- RVMA_ERR_UNIMPLEMENTED : RVMA unimplemented function
- RVMA_ERR_PROTOCOL : RVMA low-level protocol error
- RVMA_ERR_NO_EPOCH : No RVMA wait epoch found

RETURN VALUE

A read-only string describing the error.

SEE ALSO

RVMA(3)

Glossary

Definitions of common abbreviations used throughout this document.

AM	See Aperture Manager.
Aperture	Hardware within the Z bridge which maps Logical Z Addresses to SoC physical addresses.
Aperture Manager	Code within the Linux Kernel responsible for managing the Z to P aperture mapping.
Authenticatoin Manager	Sofware running on the ToRMS which is in charge of a database of identities within the machine.
Authorization Manager	A ToRMS service responsible for generating tokens describing permissions for accessing services within the machine. Particularly useful with RESTful interfaces.
Book	An 8 GB chunk of FAM. The primitive allocation unit of memory as this is the access control unit for the Firewall
Booklet	A 64 kB chunk of FAM. This amount of memory may be separately mapped through the Aperture from a book
DAX	A Linux Kernel helper module which performs operations necessary to support file systems on top of directly accessed memory.
FAM	See Fabric Attached Memory.
Fabric Attached Memory	Any memory attached through the Z bridge and accessible to all SoCs within the Local Storage Domain.
FC	See Firewall Controller.
Firewall	Hardware within the Z-bridge which controls access to FAM from the SoC. The Firewall provides separate read/write access control to each book.
Firewall Controller	Software running in Trust Zone which operates under the control of the Librarian to configure the Firewall for secure access to books within the FAM.
Firewall Proxy	Software running in Linux which relays communication between the Librarian and the Firewall Controller.
IG	See Interleave Group.

IM	See Identity Manager.
Interleave Group	The set of FAM from one or more nodes which form the smallest unit of hardware from which books may be assigned. This should be the same as the FAM contained on a single node, although the hardware allows for different configurations.
Librarian	Software running on the ToRMS which controls allocation and access to FAM for all of the SoCs.
Library File System	A Linux Kernel file system which exposes the shelves of the librarian through POSIX file APIs. It is conventionally mounted as */lfs* on each SoC.
Load/Store Domain	The collection of SoCs which can address the same collection of FAM.
LSD	See Load/Store Domain.
Shelf	A collection of books within FAM. Shelves are the application-visible allocation mechanism for managing FAM.

Index