IDT RapidIO Driver and Software Training

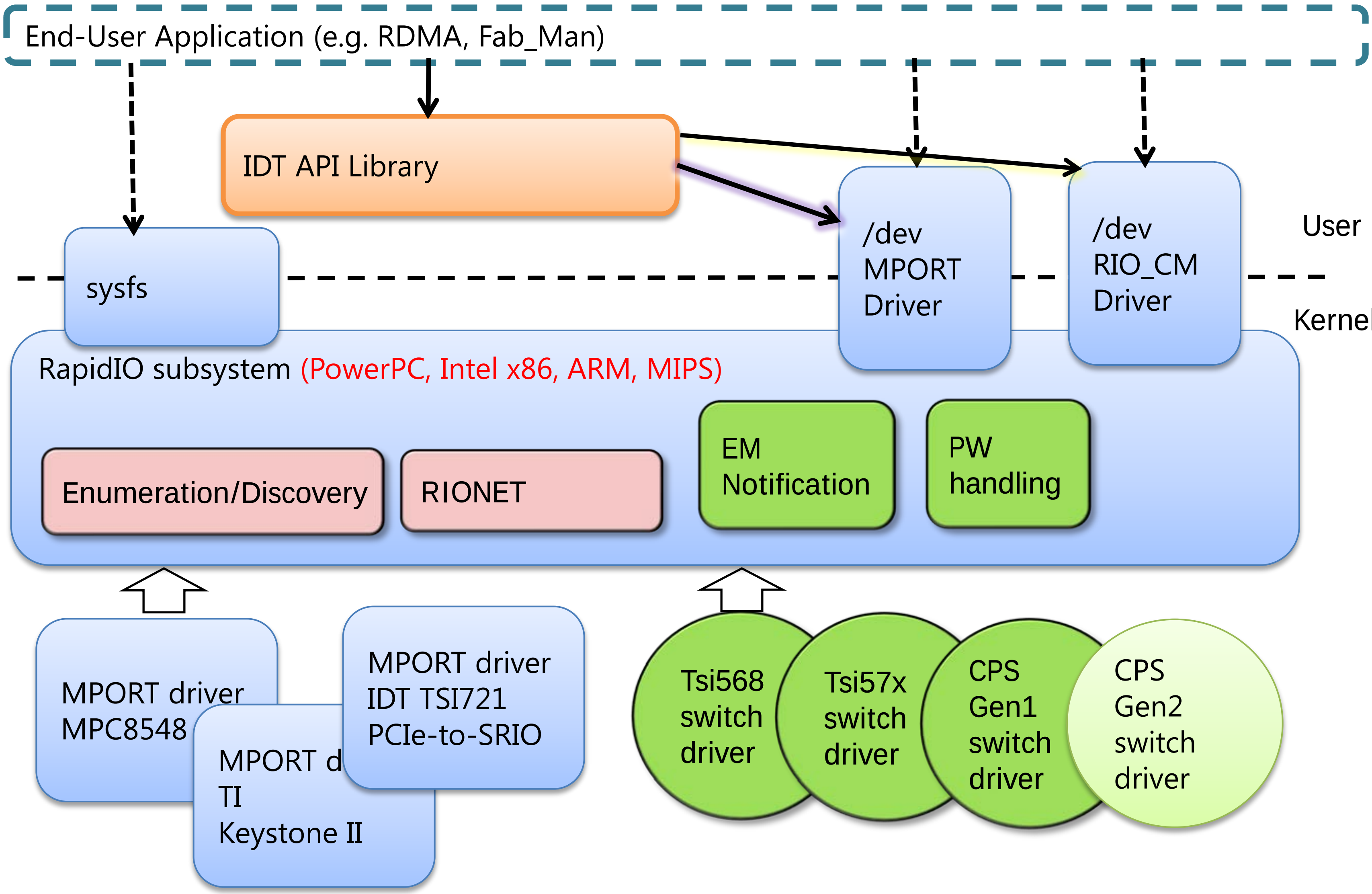
December 23, 2015

Integrated Device Technology, Inc.

Advanced Product and Technology Information
(subject to change)

CORE AND BELOW

Linux Kernel RapidIO Subsystem



Layered Design Components

- Subsystem core – defines hardware-independent kernel-space interfaces for use by other subsystem components (RIONET, enumeration, mport_cdev driver etc.). This layer allows programming in generic RapidIO terms without knowing details about HW that services requests.
- MPORT device drivers – hardware specific device drivers that provide set of callbacks defined by RapidIO subsystem. Hardware-specific MPORT device drivers are different from commonly known driver types: block and char. Consider them as plug-ins for the core.
- RapidIO switch device drivers – another subsystem/HW-specific elements with their own set of callback routines.

RIO Core to MPORT Connection

- For each MPORT device RIO core creates an object (defined as data structure rio_mport in include/linux/rio.h) which allows unified representation of different HW controllers.
- One of members of rio_mport structure is a pointer to rio_ops data structure (see include /linux/rio.h) which is created by each specific MPORT driver during registration with RIO core.
- Core interface functions that need to perform specific RapidIO operation call corresponding callback functions provided for given rio_mport object: code fragment from drivers/rapidio/rio-access.c

```
int rio_mport_read_config_##size                                          \
        (struct rio_mport *mport, u16 destid, u8 hopcount, u32 offset, type *value)  \
{                                                                         \
        int res;                                                          \
        unsigned long flags;                                              \
        u32 data = 0;                                                     \
        if (RIO_##size##_BAD) return RIO_BAD_SIZE;                        \
        spin_lock_irqsave(&rio_config_lock, flags);                       \
        res = mport->ops->cread(mport, mport->id, destid, hopcount, offset, len, &data); \
        *value = (type)data;                                              \
        spin_unlock_irqrestore(&rio_config_lock, flags);                  \
        return res;                                                       \
}
```

rio_mport Data Structure

LXR linux/include/linux/rio.h

https://lxr.missinglinkelectronics.com/linux/include/linux/rio.h

```
 * @node: Node in global list of master ports
 * @nnode: Node in network list of master ports
 * @iores: I/O mem resource that this master port interface owns
 * @riores: RIO resources that this master port interfaces owns
 * @inb_msg: RIO inbound message event descriptors
 * @outb_msg: RIO outbound message event descriptors
 * @host_deviceid: Host device ID associated with this master port
 * @ops: configuration space functions
 * @id: Port ID, unique among all ports
 * @index: Port index, unique among all port interfaces of the same type
 * @sys_size: RapidIO common transport system size
 * @phy_type: RapidIO phy type
 * @phys_efptr: RIO port extended features pointer
 * @name: Port name string
 * @dev: device structure associated with an mport
 * @priv: Master port private data
 * @dma: DMA device associated with mport
 * @nscan: RapidIO network enumeration/discovery operations
 */
struct rio_mport {
        struct list_head dbells;        /* list of doorbell events */
        struct list_head node;  /* node in global list of ports */
        struct list_head nnode; /* node in net list of ports */
        struct resource iores;
        struct resource riores[RIO_MAX_MPORT_RESOURCES];
        struct rio_msg inb_msg[RIO_MAX_MBOX];
        struct rio_msg outb_msg[RIO_MAX_MBOX];
        int host_deviceid;      /* Host device ID */
        struct rio_ops *ops;    /* low-level architecture-dependent routines */
        unsigned char id;       /* port ID, unique among all ports */
        unsigned char index;    /* port index, unique among all port
                                   interfaces of the same type */
        unsigned int sys_size;  /* RapidIO common transport system size.
                                 * 0 - Small size. 256 devices.
                                 * 1 - Large size, 65536 devices.
                                 */
        enum rio_phy_type phy_type;     /* RapidIO phy type */
        u32 phys_efptr;
        unsigned char name[RIO_MAX_MPORT_NAME];
        struct device dev;
        void *priv;             /* Master port private data */
#ifdef CONFIG_RAPIDIO_DMA_ENGINE
        struct dma_device       dma;
#endif
        struct rio_scan *nscan;
};

/*
```

rio_ops Data Structure

LXR linux/include/linux/rio.h

https://lxr.missinglinkelectronics.com/linux/include/linux/rio.h

Apps Yahoo! Linux Weather Google Dictionary tsi721 OFED Git - Reference PortServer TS 16 WTI - Power Switch RapidIO: STG GitHub Other bookmarks

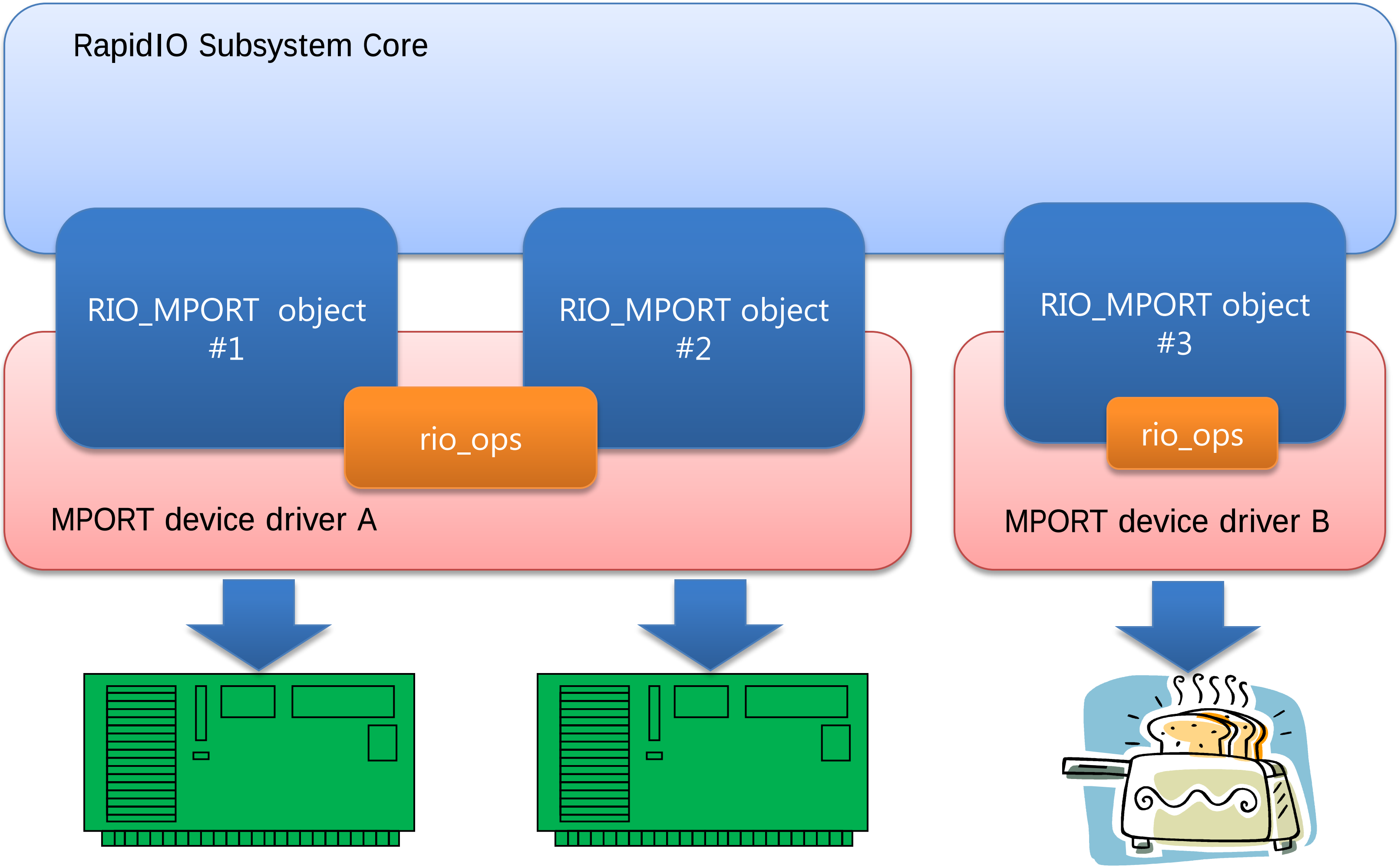
```
/**
 * struct rio_ops - Low-level RIO configuration space operations
 * @lcread: Callback to perform local (master port) read of config space.
 * @lcwrite: Callback to perform local (master port) write of config space.
 * @cread: Callback to perform network read of config space.
 * @cwrite: Callback to perform network write of config space.
 * @dsend: Callback to send a doorbell message.
 * @pwenable: Callback to enable/disable port-write message handling.
 * @open_outb_mbox: Callback to initialize outbound mailbox.
 * @close_outb_mbox: Callback to shut down outbound mailbox.
 * @open_inb_mbox: Callback to initialize inbound mailbox.
 * @close_inb_mbox: Callback to shut down inbound mailbox.
 * @add_outb_message: Callback to add a message to an outbound mailbox queue.
 * @add_inb_buffer: Callback to add a buffer to an inbound mailbox queue.
 * @get_inb_message: Callback to get a message from an inbound mailbox queue.
 * @map_inb: Callback to map RapidIO address region into local memory space.
 * @unmap_inb: Callback to unmap RapidIO address region mapped with map_inb().
 */
struct rio_ops {
        int (*lcread) (struct rio_mport *mport, int index, u32 offset, int len,
                        u32 *data);
        int (*lcwrite) (struct rio_mport *mport, int index, u32 offset, int len,
                        u32 data);
        int (*cread) (struct rio_mport *mport, int index, u16 destid,
                        u8 hopcount, u32 offset, int len, u32 *data);
        int (*cwrite) (struct rio_mport *mport, int index, u16 destid,
                        u8 hopcount, u32 offset, int len, u32 data);
        int (*dsend) (struct rio_mport *mport, int index, u16 destid, u16 data);
        int (*pwenable) (struct rio_mport *mport, int enable);
        int (*open_outb_mbox)(struct rio_mport *mport, void *dev_id,
                              int mbox, int entries);
        void (*close_outb_mbox)(struct rio_mport *mport, int mbox);
        int  (*open_inb_mbox)(struct rio_mport *mport, void *dev_id,
                             int mbox, int entries);
        void (*close_inb_mbox)(struct rio_mport *mport, int mbox);
        int  (*add_outb_message)(struct rio_mport *mport, struct rio_dev *rdev,
                                 int mbox, void *buffer, size_t len);
        int (*add_inb_buffer)(struct rio_mport *mport, int mbox, void *buf);
        void *(*get_inb_message)(struct rio_mport *mport, int mbox);
        int (*map_inb)(struct rio_mport *mport, dma_addr_t lstart,
                        u64 rstart, u32 size, u32 flags);
        void (*unmap_inb)(struct rio_mport *mport, dma_addr_t lstart);
};

#define RIO_RESOURCE_MEM        0x00000100
#define RIO_RESOURCE_DOORBELL   0x00000200
#define RIO_RESOURCE_MAILBOX    0x00000400
```

How it works together

- ❑ During driver initialization an MPORT driver registers its rio_mport object (including supported operations) within the RIO core.
- ❑ RIO core uses rio_mport object as an abstract representation of HW device during its operations.
- ❑ When RIO core function needs to perform operation controlled by HW (e.g. maintenance read from remote target device) it calls corresponding function supplied through the rio_mport object.

- ➢ If a RIO operation is not supported by given mport device, pointer to that operation in rio_ops must to be set to NULL. Some operations are mandatory for implementation and RIO core does not check for NULL pointer.

Example of Multiport Configuration



HW Abstraction at Work

```
rio_mport_read_config_32(port, destid, hopcount,
                         RIO_DEV_ID_CAR,
                         &result);
```

Upper level kernel module (e.g rio-scan) calls core interface function.

```
res = mport->ops->cread(mport, mport->id,
      destid, hopcount, offset, len, &data);
```

RIO core maps the call into corresponding mport operation.

```
static int tsi721_cread_dma(*mport, index,
              destid, hcount, offset, len, *data)
{
.....
return tsi721_maint_dma(priv, mport->sys_size,
              destid, hopcount,offset, len, data, 0);
}
```

MPORT device driver performs requested SRIO operation and returns a result back through the call chain.

DMA Operations

- ❑ Support for DMA data transfers is different from callback scheme described earlier.
- ❑ DMA operations in Linux kernel must be provided through generic DMA engine framework.
- ❑ MPORT device drivers that support DMA implement and register their DMA operations with Linux kernel DMA engine.
- ❑ RapidIO uses SLAVE DMA mode defined by DMA engine with subsystem specific extension.
- ❑ To communicate with DMA engine RIO core provides interface functions:
  - rio_request_mport_dma(struct rio_mport *mport)
  - rio_request_dma(struct rio_dev *rdev)
  - rio_release_dma(struct dma_chan *dchan)
  - rio_dma_prep_xfer(struct dma_chan *dchan, ...)

DMA Operations (cont.)

- ❑ Functions listed above address only RIO-specific part of DMA transfers programming.
- ❑ After RIO-capable DMA channel have been allocated standard DMA engine operations should be used.
- ❑ Not every RIO controller provides HW DMA support for data transfers to/from remote RIO target device.
- ❑ Even HW DMA support exists it does not guarantee that DMA engine support have been implemented for that device/mode. E.g. kernel DMA engine code for Freescale's MPC8548 device supports memory-to-memory and regular SLAVE mode, but RIO-compatible SLAVE mode is not implemented (yet).

Different RapidIO Controllers

❑ HW controllers supported by kernel RapidIO subsystem can be divided into two groups:

- Built-in SRIO controllers inside of SoC (Freescale, TI, Cavium, FPGAs, etc.).
- External bridges to SRIO from busses like PCI or PCI Express (PCI - Tsi620, PCIe – Tsi721).

❑ Built-in controllers are specific for given SoC architecture and their mport drivers are located in corresponding arch code branches (this is why we do not see them in “drivers/rapidio” directory).

❑ MPORT drivers for external bridges are (mostly) architecture independent and are locates in “drivers/rapidio” directory.

❑ Both forms of mport drivers register their rio_mport object with RIO subsystem core and therefore their operations are transparent to callers of core interface functions.

Different RapidIO Controllers (ii)

SRIO controllers/bridges do not provide an equal level of RapidIO functionality. Currently only Tsi721 provides all set of functions (options) supported by RapidIO subsystem core. Here are examples of some HW or SW limitations:

- Freescale SoCs – have reduced number of RIO messaging MBOXes, do not have DMA engine support for RIO (HW is OK).
- TI Keystone II SoC – does not have inbound window address translation capability.

PCIe vs. SRIO (SW POV)

- Different enumeration perspective:
  - PCIe is single-root tree (many RCs are possible) with address-based routing.
    - Address-based routing can be a limiting factor in large systems.
    - PCIe eliminated peer-to-peer capability that was available in older PCI implementations.
  - SRIO is fabric with possibility of multipath routing and redundant mport connections on the same node.

ABOVE THE CORE

Kernel Users of RapidIO Core

- RapidIO subsystem has multiple components that act as users of generic interface functions exported by the core.
- Those functional components are hardware independent and can work on any platform that support necessary RapidIO operations.
- Upper layer kernel RapidIO subsystem components:
  - Enumeration/Discovery (RIO_SCAN) – implements a basic RapidIO fabric enumeration/discovery process which automatically assigns destination IDs to endpoints and programs switch routing tables.
  - RIONET - Ethernet emulation over RapidIO. This driver allows to use RapidIO controllers as common Ethernet NIC with all available TCP/IP stack support.
  - SYSFS support – RapidIO subsystem creates its own sysfs entries that allow to analyze state of the subsystem.
  - MPORT_CDEV – universal character mode device driver that allows user-space communications with RapidIO fabric.

Enumeration Variants

- To make RapidIO devices visible/accessible to node users SRIO fabric has to be enumerated.
- Compared to PCI/PCIe bus enumeration, SRIO fabric enumeration may have multiple methods depending on system configuration and use model.
- Any RapidIO system can implement its own enumeration method.
- As an example of enumeration/discovery process Linux kernel RapidIO subsystem includes a basic implementation which automatically assigns destination IDs to endpoints and programs switch routing tables (based on Annex 1 of RapidIO specification):
  - Enumeration is performed by a dedicated node (host). RapidIO enumerating host is defined by kernel boot command line parameter “rapidio.hdid=n,...”.
  - All other nodes (targets) in RapidIO network perform the discovery process.

Enumeration Variants (ii)

- The basic implementation can enumerate the RIO fabric only in "fresh after power-on" state (all nodes). It does not support re-enumeration of the fabric or hot-plug node/segment addition/removal.
- The basic enumeration/discovery process is implemented as a loadable kernel module (rio-scan.c) that can be replaced by an end-user if required.
- RIO subsystem core offers mechanism to register enumerating process on per-mport basis. Therefore, if required users can implement different enumeration/discovery processes for each mport.
- With MPORT_CDEV device driver availability users have got ability to implement their own enumeration algorithms as user space applications instead of kernel module.

RIO_MPORT_CDEV DEVICE DRIVER

RIO_MPORT_CDEV Driver

- ❑ This driver allows to perform RIO operations from user space applications.
- ❑ It implements common IOCTL interfaces between kernel RIO subsystem core and DMA engine and user space.
- ❑ Next slide demonstrates request path from user space to SRIO bus.

RIO_MPORT_CDEV Driver (ii)

```
ioctl(hnd->fd, IO_MPORT_MAINT_READ_REMOTE, &mt)
```

User-space application or library issues IOCTL request to mport_cdev device driver.

```
rio_mport_read_config_32(port, destid, hopcount,
                         RIO_DEV_ID_CAR,
                         &result);
```

mport_cdev processes IOCTL request and calls core interface function.

```
res = mport->ops->cread(mport, mport->id,
  destid, hopcount, offset, len, &data);
```

RIO core maps the call into corresponding mport operation.

MPORT device driver performs requested SRIO operation and returns a result back through the call chain (see slide 11).

Debugging Options

- RapidIO subsystem offers several debug tracing options that allow end user to analyze component operations for better understanding or issue resolution.
- At the top level debug traces for RapidIO subsystem are controlled by kernel configuration option CONFIG_RAPIDIO_DEBUG.
- CONFIG_RAPIDIO_DEBUG enables debug output from all subsystem components.
- To reduce amount of produced debug information some newer components implement filtered debug output options controlled through kernel module parameters.
- Currently this option available for Tsi721 mport driver, rio_mport_cdev driver and rio_cm driver.
- Older subsystem components will be converted to support this option as time permits.

Debugging tsi721_mport driver

- ❑ Debugging output from this driver is controlled by kernel module parameter “dbg_level”.
- ❑ By default this parameter is set to 0, what means “disabled”.
- ❑ Two ways to set this parameter (values are for example only):
  - When loading the module: “modprobe tsi721_mport dbg_level=0x23”, or
  - at runtime using sysfs (with root privileges):

    “echo 0x1f > /sys/module/tsi721_mport/parameters/dbg_level”
- ❑ Current value of dbg_level can be displayed at any time by entering the following command: “cat /sys/module/tsi721_mport/parameters/dbg_level”. NOTE: value is returned in decimal format, convert to hex to match the mask.
- ❑ Debug output bit masks are listed below (see drivers/rapidio/devices/tsi721.h for updates)

```
DBG_INIT= 0x01      /* driver initialization */
DBG_EXIT= 0x02      /* driver exit/removal */
DBG_MPORT= 0x04 /* mport add/remove */
DBG_MAINT= 0x08  /* maintenance ops messages */
DBG_DMA= 0x10      /* DMA transfer messages */
DBG_DMAV= 0x20    /* verbose DMA transfer messages */
DBG_IBW= 0x40      /* inbound window messages */
DBG_EVENT= 0x80  /* event handling messages */
DBG_OBW= 0x100   /* outbound window messages */
DBG_DBELL=0x200  /* doorbell configuration/handling messages */
DBG_OMSG=0x400  /* outbound MBOX messages */
DBG_IMSG= 0x800  /* inbound MBOX messages */
```

Debugging rio_mport_cdev driver

- ❑ Debugging output from this driver is controlled by kernel module parameter "dbg_level".
- ❑ By default this parameter is set to 0, what means "disabled".
- ❑ Two ways to set this parameter (values are for example only):
  - When loading the module: "modprobe rio_mport_cdev dbg_level=0x23", or
  - at runtime using sysfs (with root privileges):
    "echo 0x1f > /sys/module/rio_mport_cdev/parameters/dbg_level"
- ❑ Current value of dbg_level can be displayed at any time by entering the following command: "cat /sys/module/rio_mport_cdev/parameters/dbg_level". NOTE: value is returned in decimal format, convert to hex to match the mask.
- ❑ Debug output bit masks are listed below (see drivers/rapidio/devices/rio_mport_cdev.c)

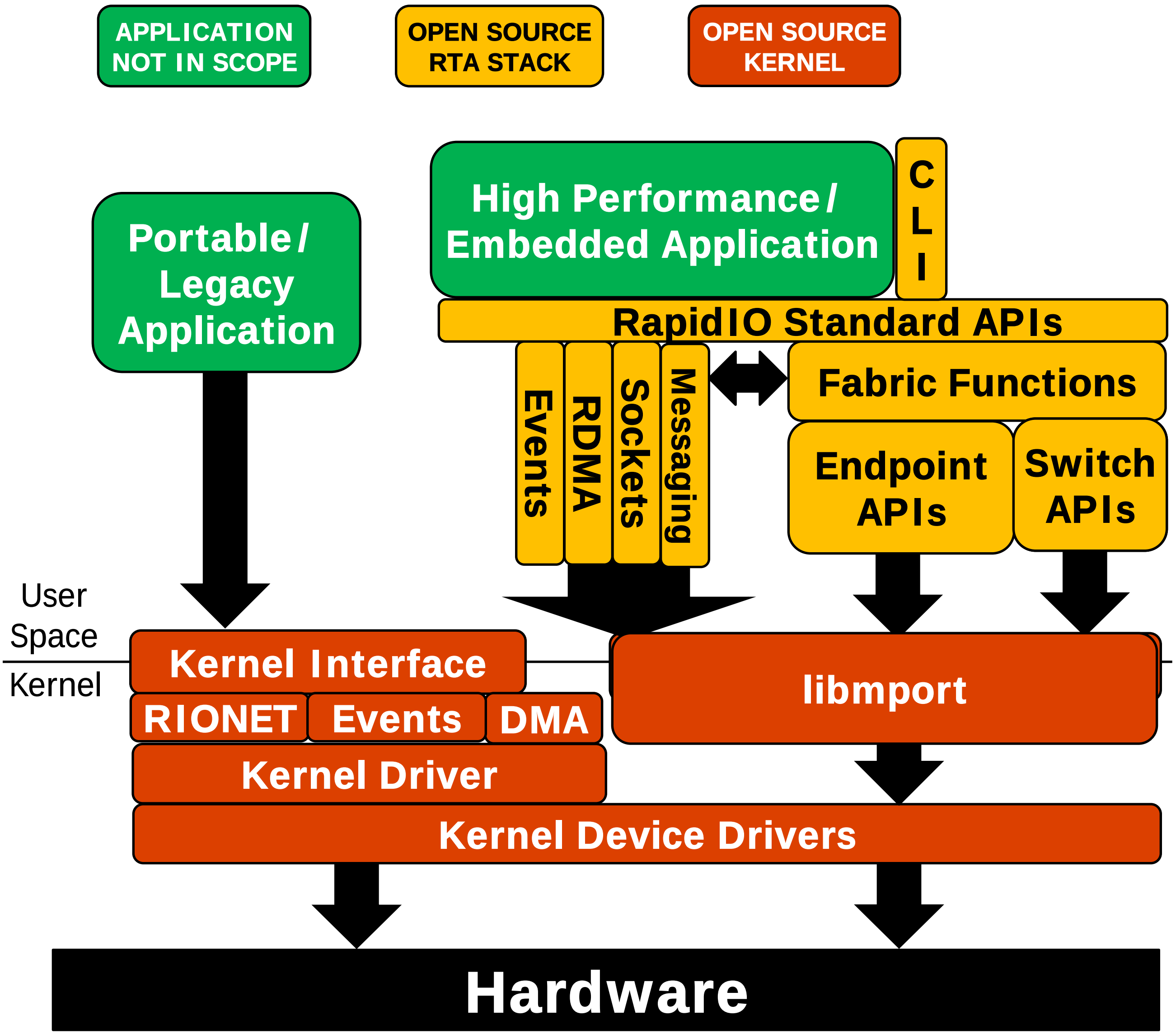
```
DBG_INIT= 0x01     /* driver init */
DBG_EXIT= 0x02     /* driver exit */
DBG_MPORT= 0x04 /* mport add/remove */
DBG_RDEV=0x08      /* RapidIO device add/remove */
DBG_DMA= 0x10      /* DMA transfer messages */
DBG_MMAP=0x20      /* mapping messages */
DBG_IBW=0x40       /* inbound window */
DBG_EVENT= 0x80  /* event handling messages */
DBG_OBW=0x100      /* outbound window messages */
DBG_DBELL=0x200  /* doorbell messages */
```

Tips for Debugging

- ❑ Masks for both drivers seems to be similar but because they are used at different layers sequential approach can be useful to reduce amount of produced messages.
- ❑ When debugging user space application start with traces from rio_mport_cdev and decide if an error is reported by this driver or reported by tsi721_mport driver.
- ❑ If needed add traces from tsi721_mport driver.

RAPIDIO SOFTWARE BACKGROUND

Conceptual - Open Source Stack



CLI
Open source command line interpreter for fabric management

RapidIO Standard APIs
RapidIO standard interface definitions and behavior

Fabric Functions
Implementation of RapidIO Standard APIs Fabric Management

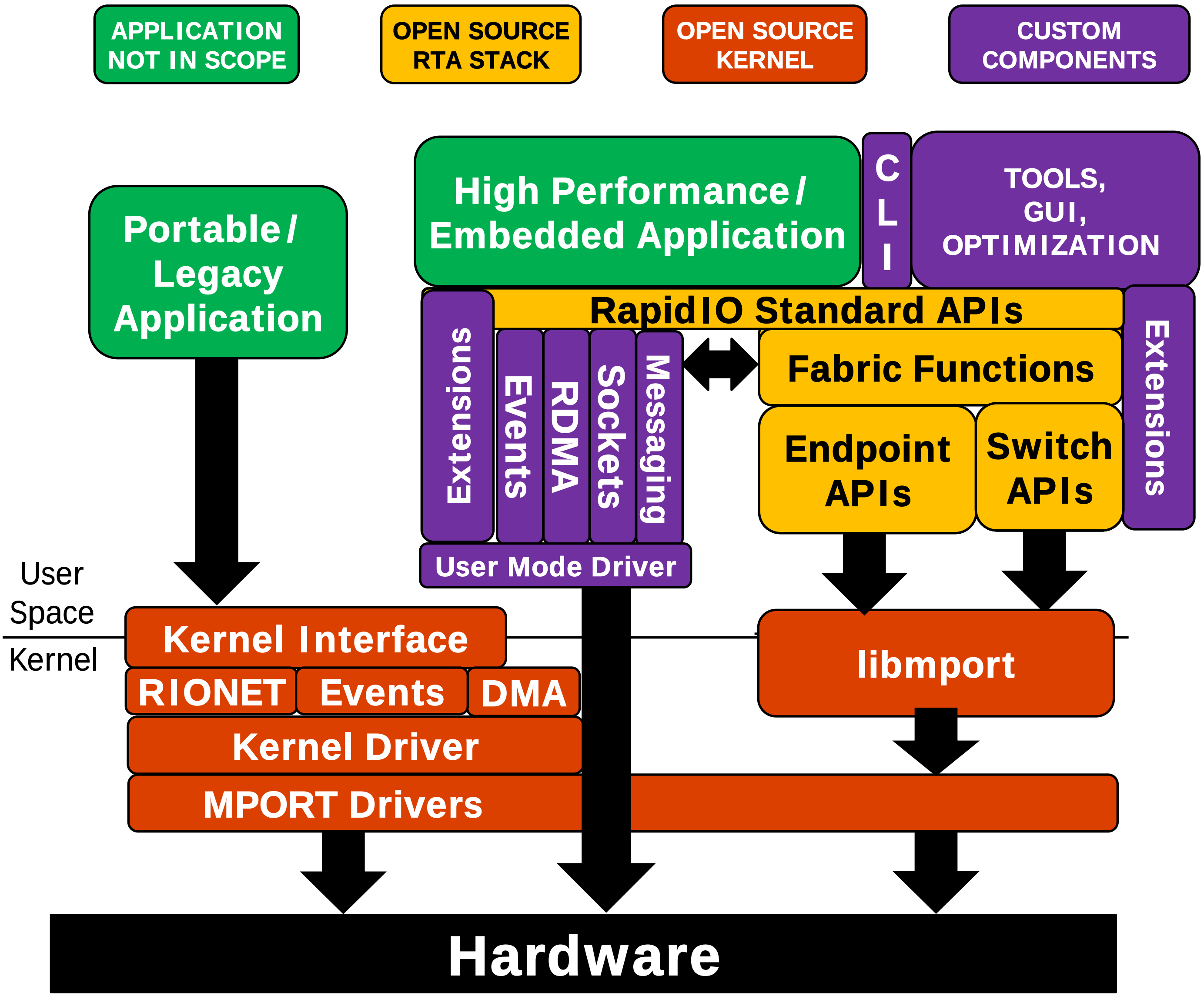
Endpoint APIs
Universal programming model for endpoint functions

Switch APIs
Universal programming model for switch functions.

Messaging/Sockets/RDMA/Events
Implementation of RapidIO Standard API's Data Path

libmport
Character mode driver, interface to kernel device drivers

Customization, Optimization



TOOLS, GUI, OPTIMIZATION
Value adders for debug/monitoring, system visualization, data interpretation, and topology specific functions

User Mode Driver
User mode drivers optimized for executing hardware operations in user mode.

Extensions
Additional data path and/or fabric management services provided to applications

Philosophy

<u>Minimum</u> functionality to demonstrate RapidIO value:

- High throughput
- Low latency
- Fault tolerant

<u>Portable</u> between operating systems, hardware configurations, and applications

<u>Interoperable</u> between any endpoint and/or switch compliant to the standard

<u>Extensible</u> to enable innovation by all members of the RapidIO community

RAPIDIO SOFTWARE STACK PROCESS STRUCTURE

Installing the RapidIO Software Stack

- Clone the rapidio_sw repository from [www.github.com/RapidIO into /opt/rapidio/](www.github.com/RapidIO)
  - Creates /opt/rapidio/rapidio_sw directory

- in the “rapidio_sw” directory, type “make all”

- create /etc/rapidio/rio.conf file to control master and slave Fabric Management Daemons
  - Examples: fabric_management/daemon/cfg
  - GRY05: Master
  - GRY06: Slave

Running the RapidIO Software Stack

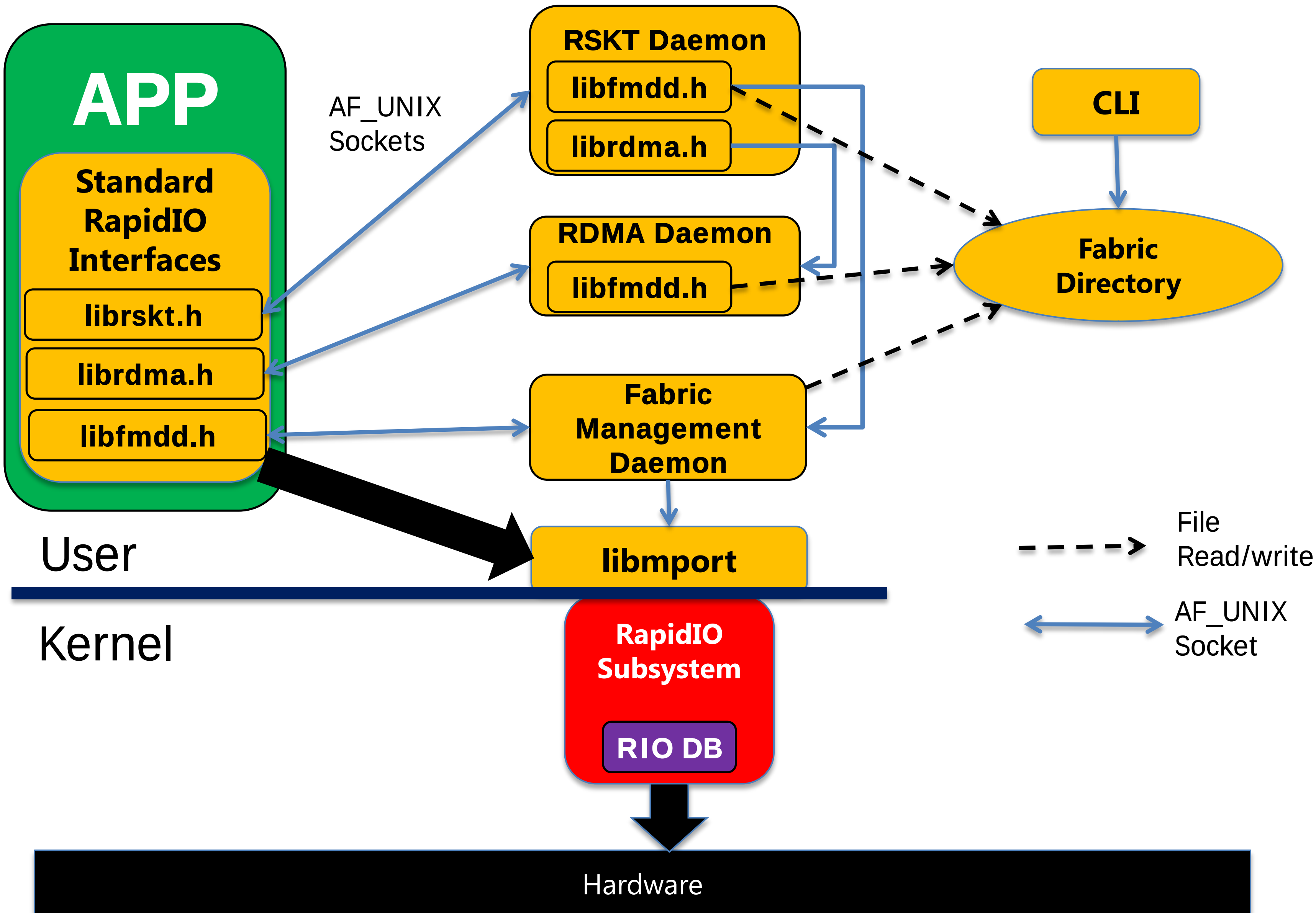
- the scripts in /opt/rapidio/ will use the executables in /opt/rapidio/rapidio_sw to start the RapidIO Software stack

- Similar to previous installation:
  - ./all_down.sh – powers down all nodes
  - ./all_start.sh – starts kernel RapidIO subsystem with kernel enumeration
  - ./ibox_start_all – starts user mode RapidIO subsystem with user mode enumeration

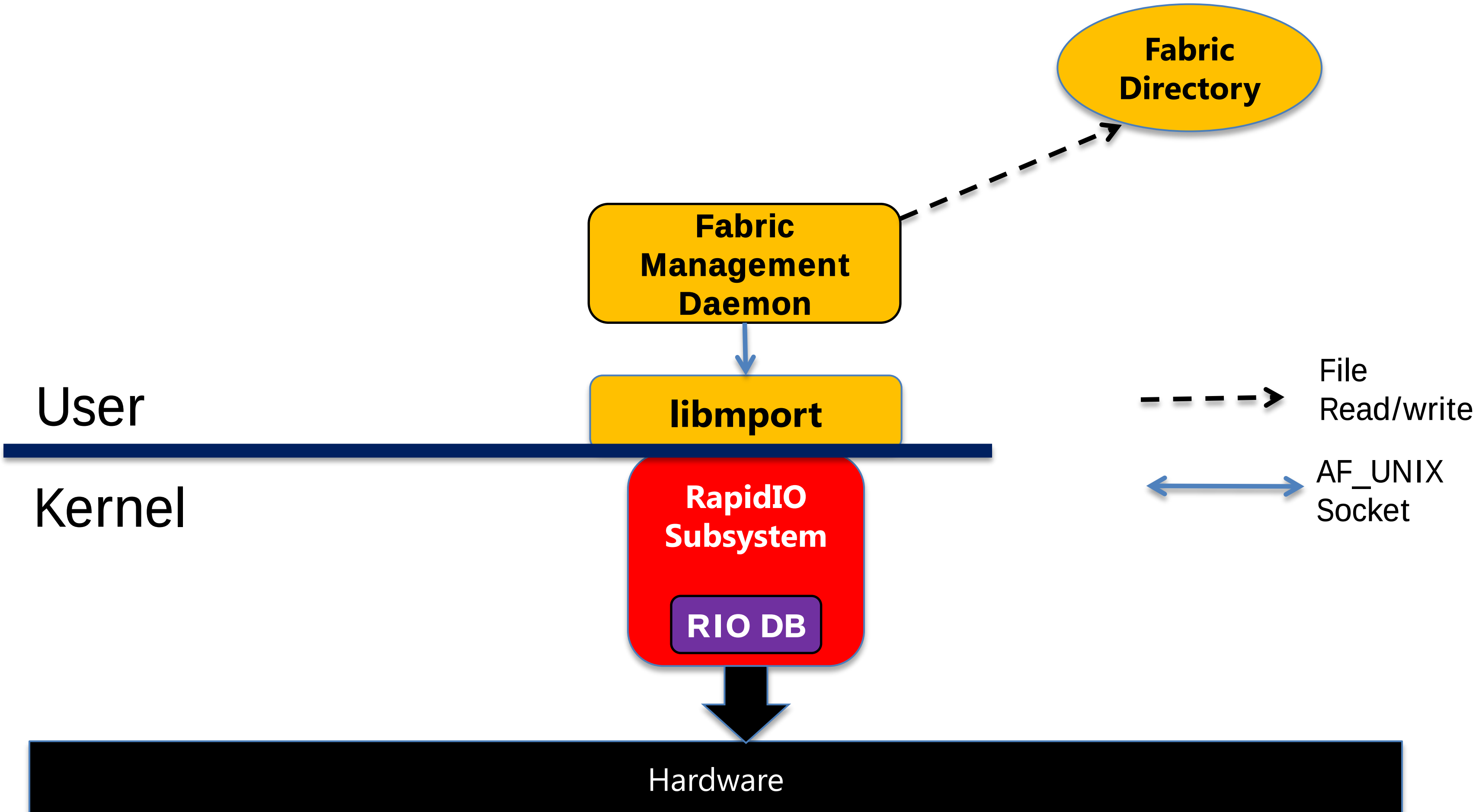
Daemons and Libraries

Function	Daemon	Implementation rapidio_sw/	Library in /include	Implementation rapidio_sw/
Fabric Management	FMD	fabric_management /daemon	libfmdd.h	fabric_management/ libfmdd.h
RDMA	RDMAD	rdma/daemon	librdma.h	rdma/lib
RDMA Sockets	RSKTD	rdma/rskt/daemon	librskt.h	rdma/rskt/libr
Command Line Interpreter	CLI	TBD	N/A	N/A
Fabric Directory	N/A	Posix Shared Memory Region	N/A	N/A

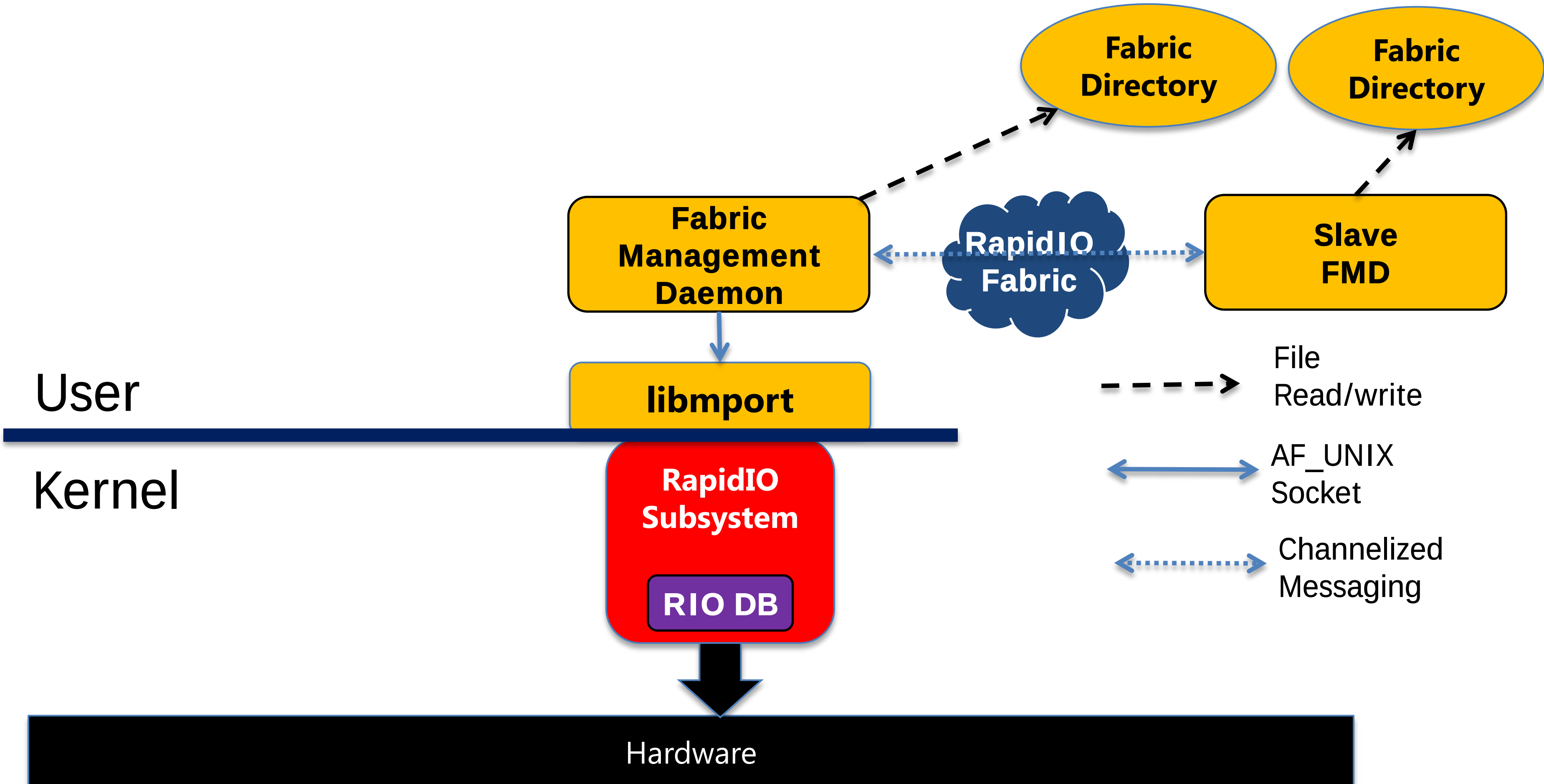
Library & Daemon Relationships



Startup – Master FMD First

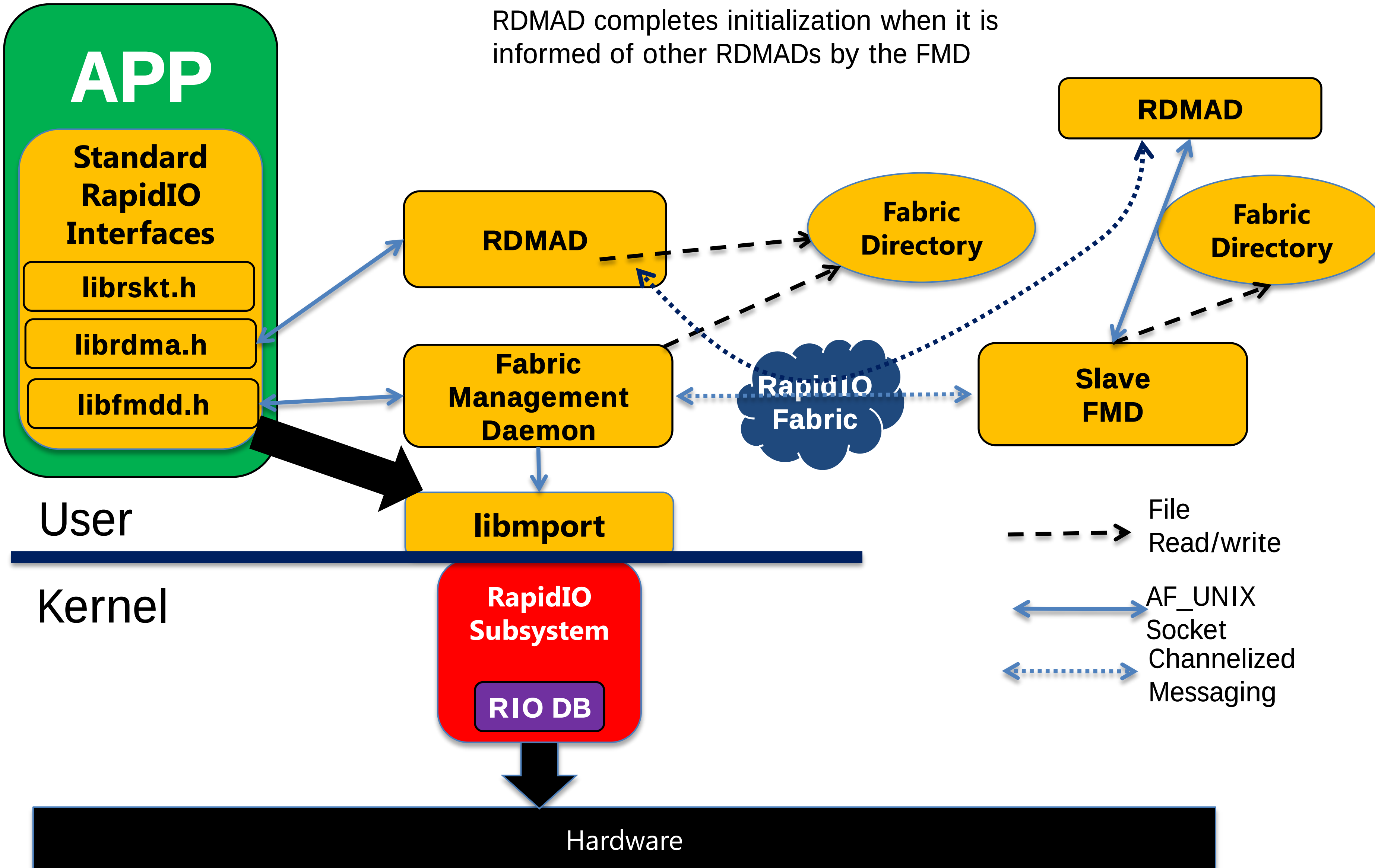


Startup – Slave FMDs Next

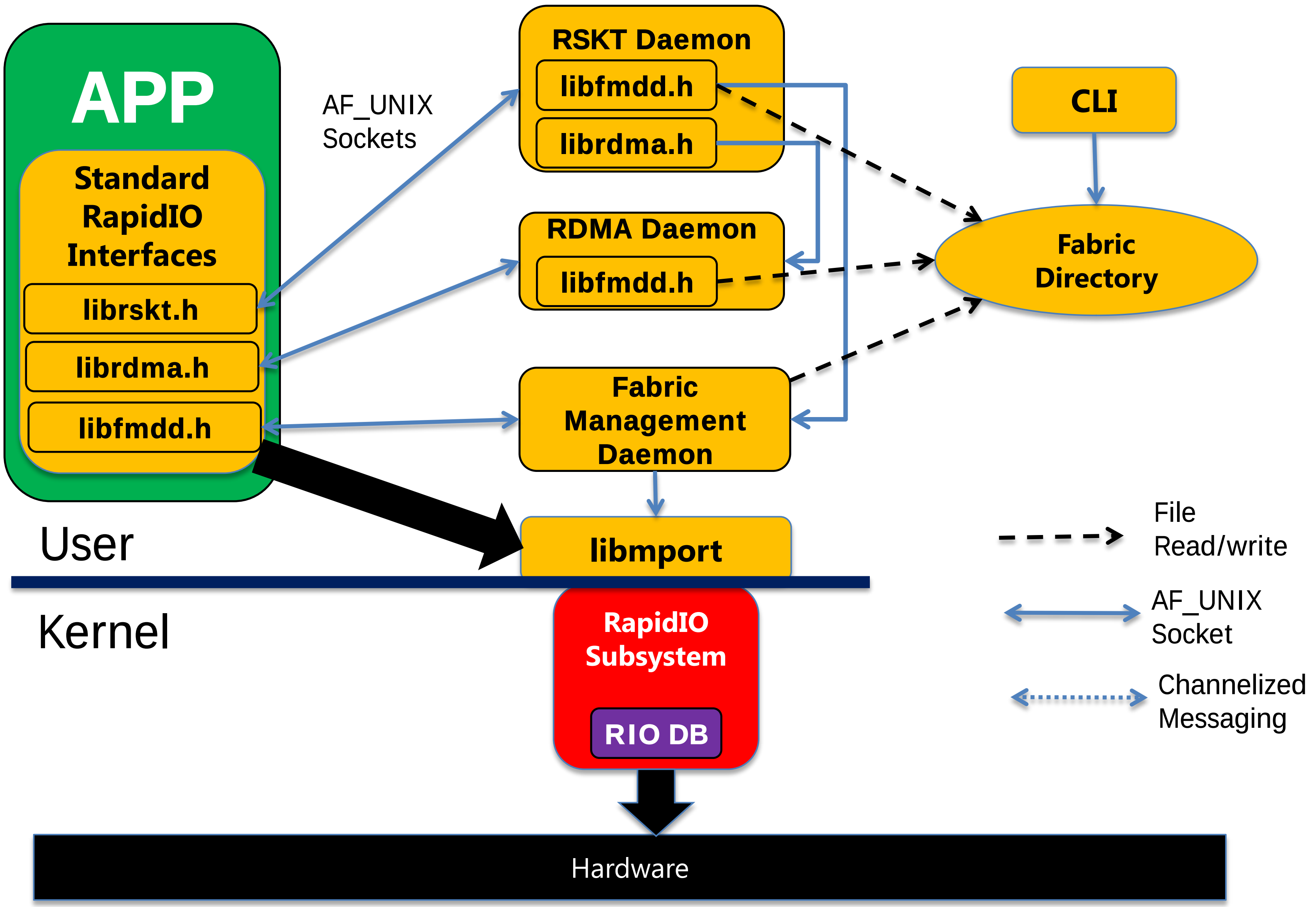


Startup – RDMADs, in any order

RDMAD completes initialization when it is informed of other RDMADs by the FMD



Startup – RSKTDs last



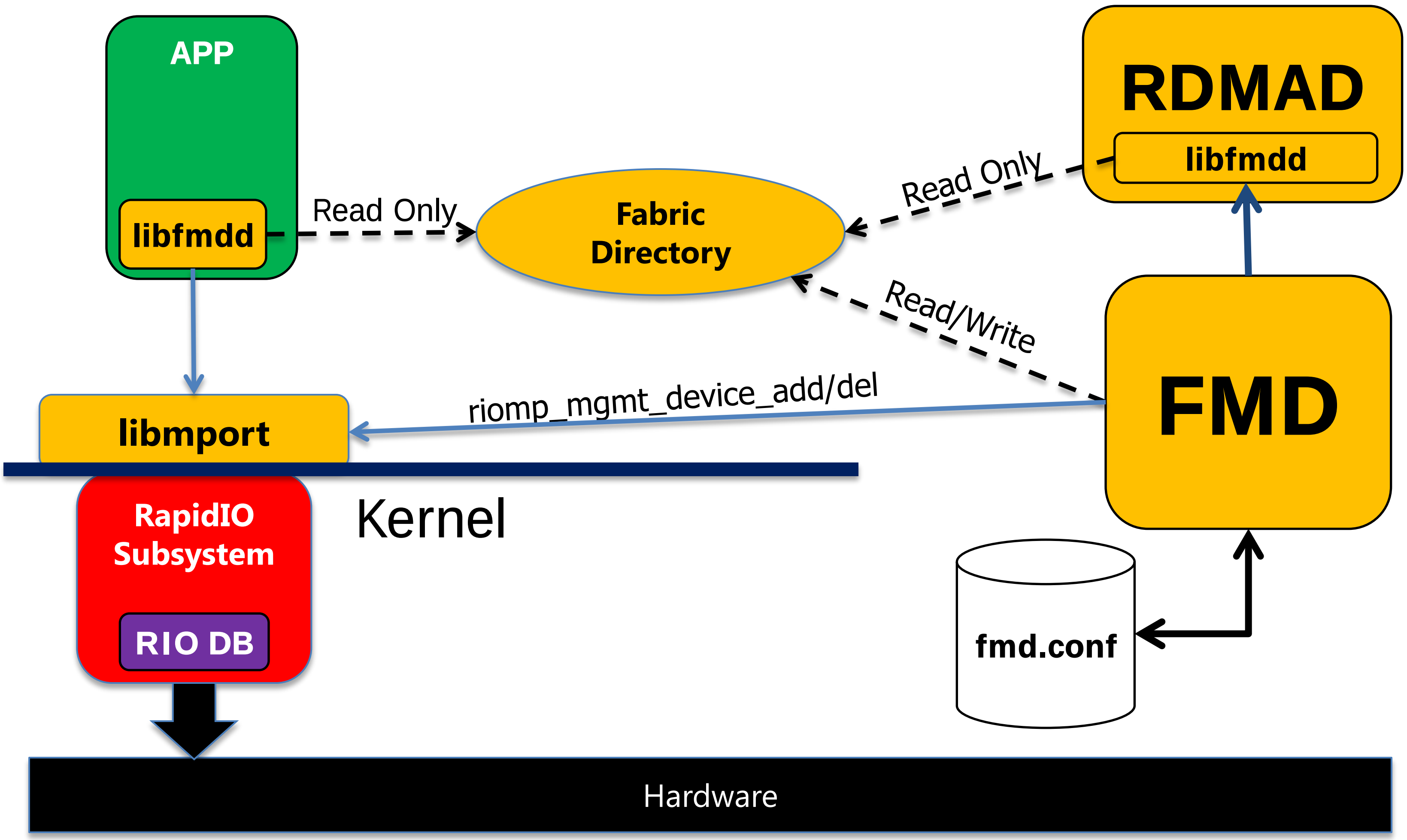
RAPIDIO FABRIC MANAGEMENT OVERVIEW

System enumeration
- uses configuration file

Notification of application availability
-Tells all nodes when RDMA, RSKT applications are available

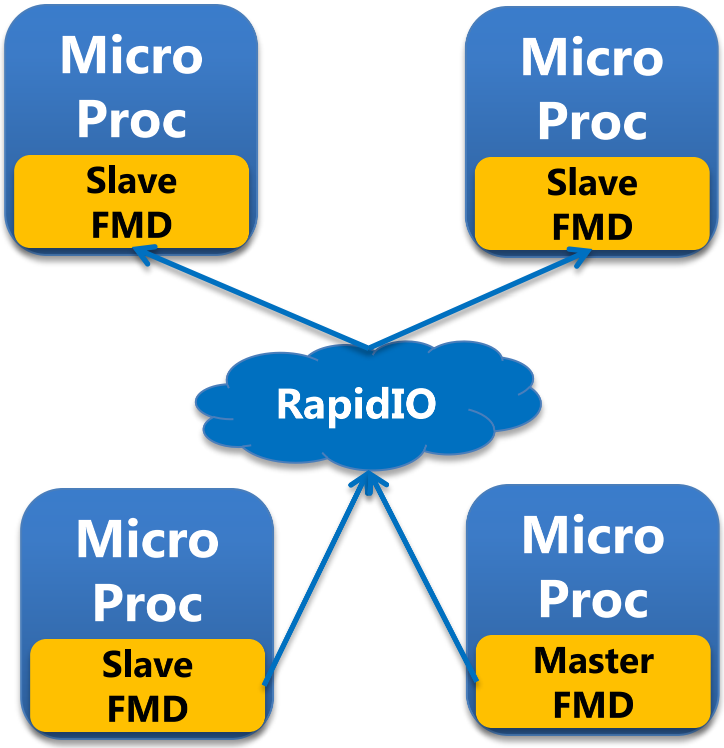
- Fault tolerance
- Supports hot swap
- hardware and software robustness

User Mode Fabric Management



RDMA Library gets status information from the Fabric Directory
Fabric Management Daemon receives all port writes
Fabric Management Daemon uses <u>libmport</u> to manage Kernel Devices

Fabric Management - Design



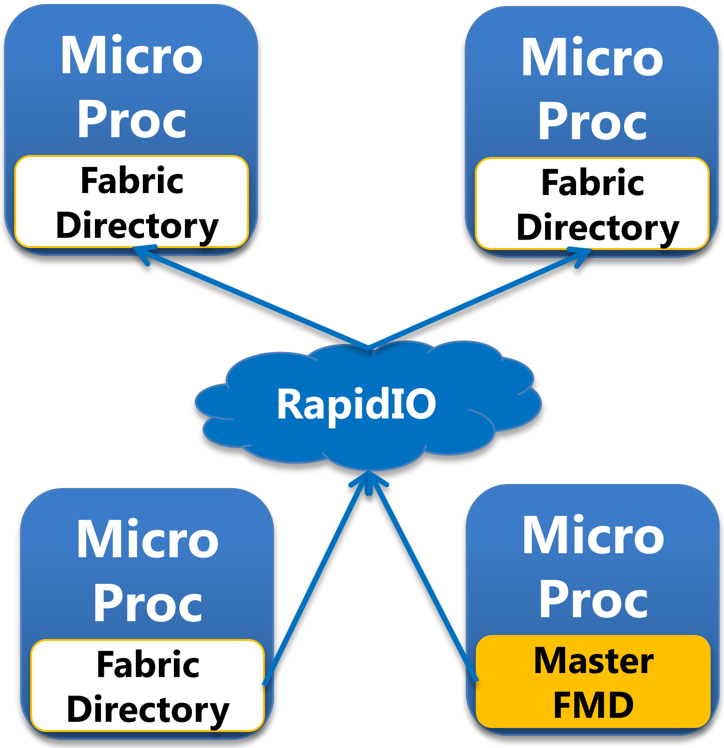
Master Fabric Management Daemon (FMD) enumerates system, then informs Slave FMDs to read configuration of the system..

Hot swap events cause port-writes to be sent to all FMDs.

Each FMD updates its local database and applications.

DISTRIBUTED FABRIC MANAGEMENT

Fabric Management - Possibilities



Master Fabric Management Daemon (FMD) pushes/writes system configuration information to RDMA memory on remote devices.

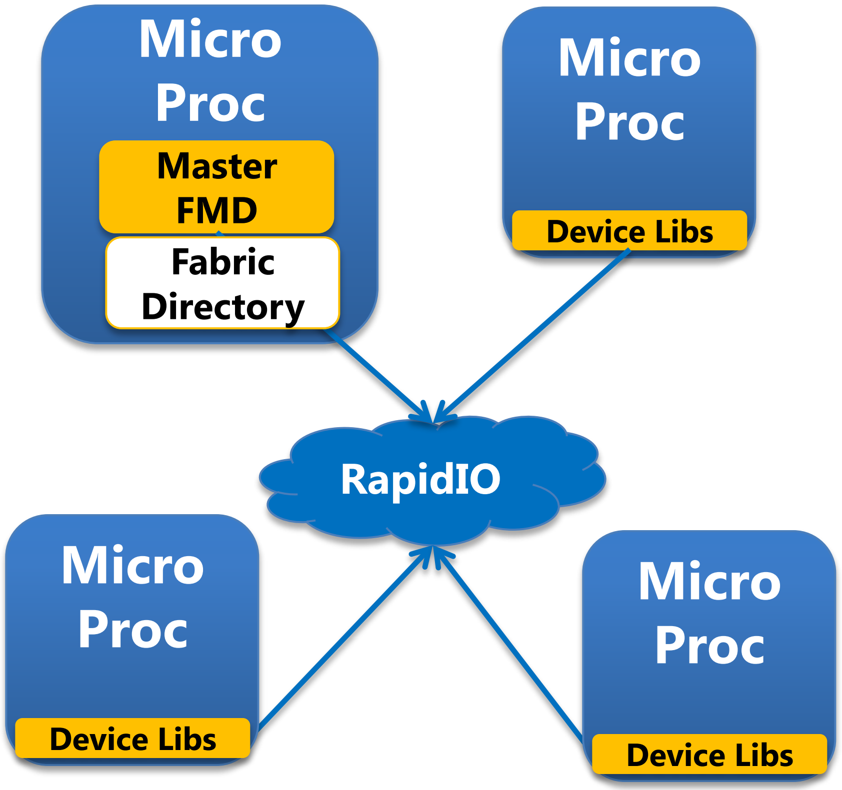
Centralizes fabric event management

Redundant Master FMD's possible

Simplifies system partitioning
-Master FMD "knows" what switches, endpoints, destination IDs it controls
- Services outside the FMD only know the devices in the Fabric Directory

CENTRALIZE FABRIC MANAGEMENT

Fabric Management - Possibilities



Services (i.e. RDMA) running on remote devices pull/read system configuration information from Master RDMA read-only accessible region

Remote access using source/target enforced read only RDMA

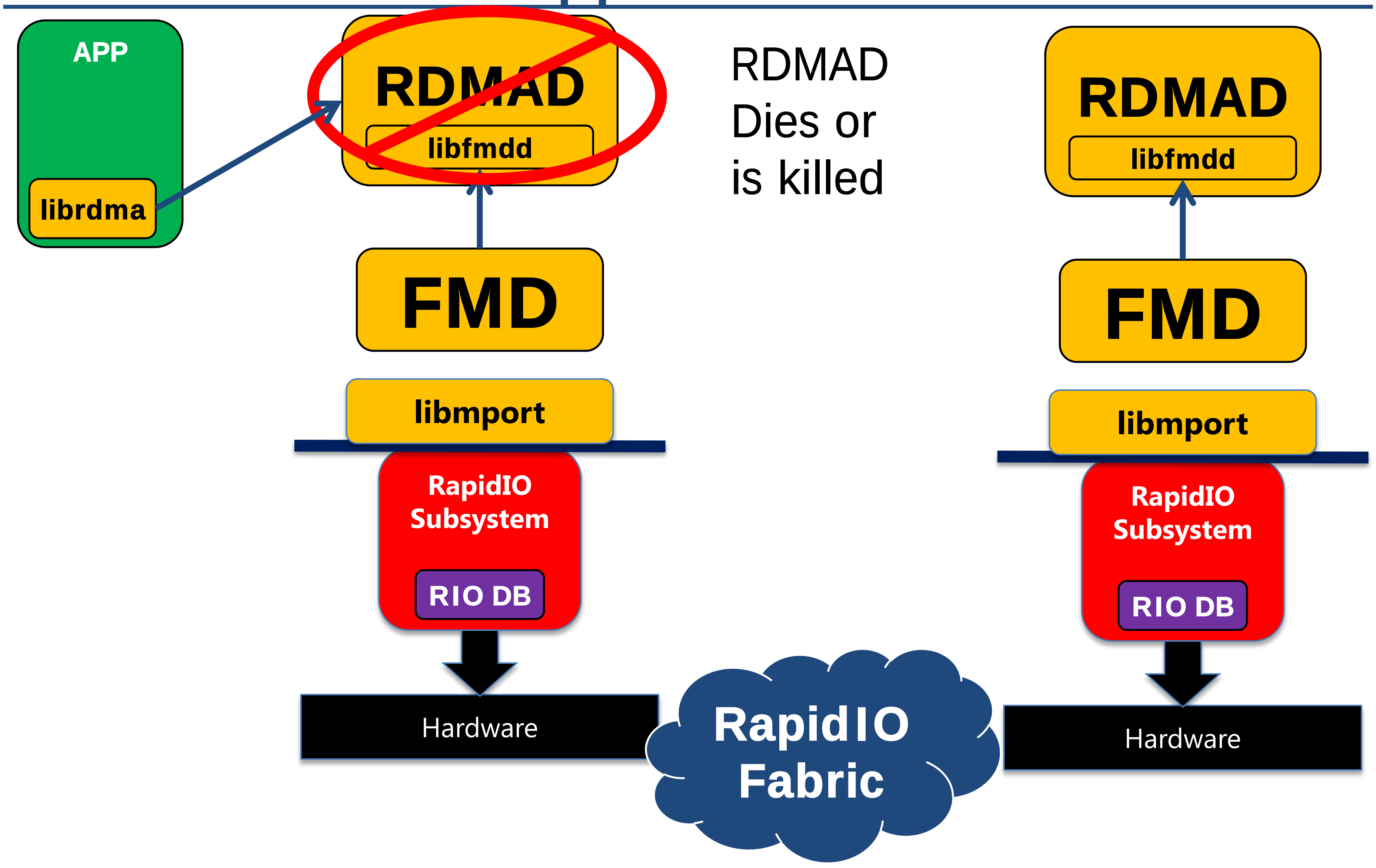
Could also use a cache coherency protocol for synchronization

Centralized, secure, scalable network management

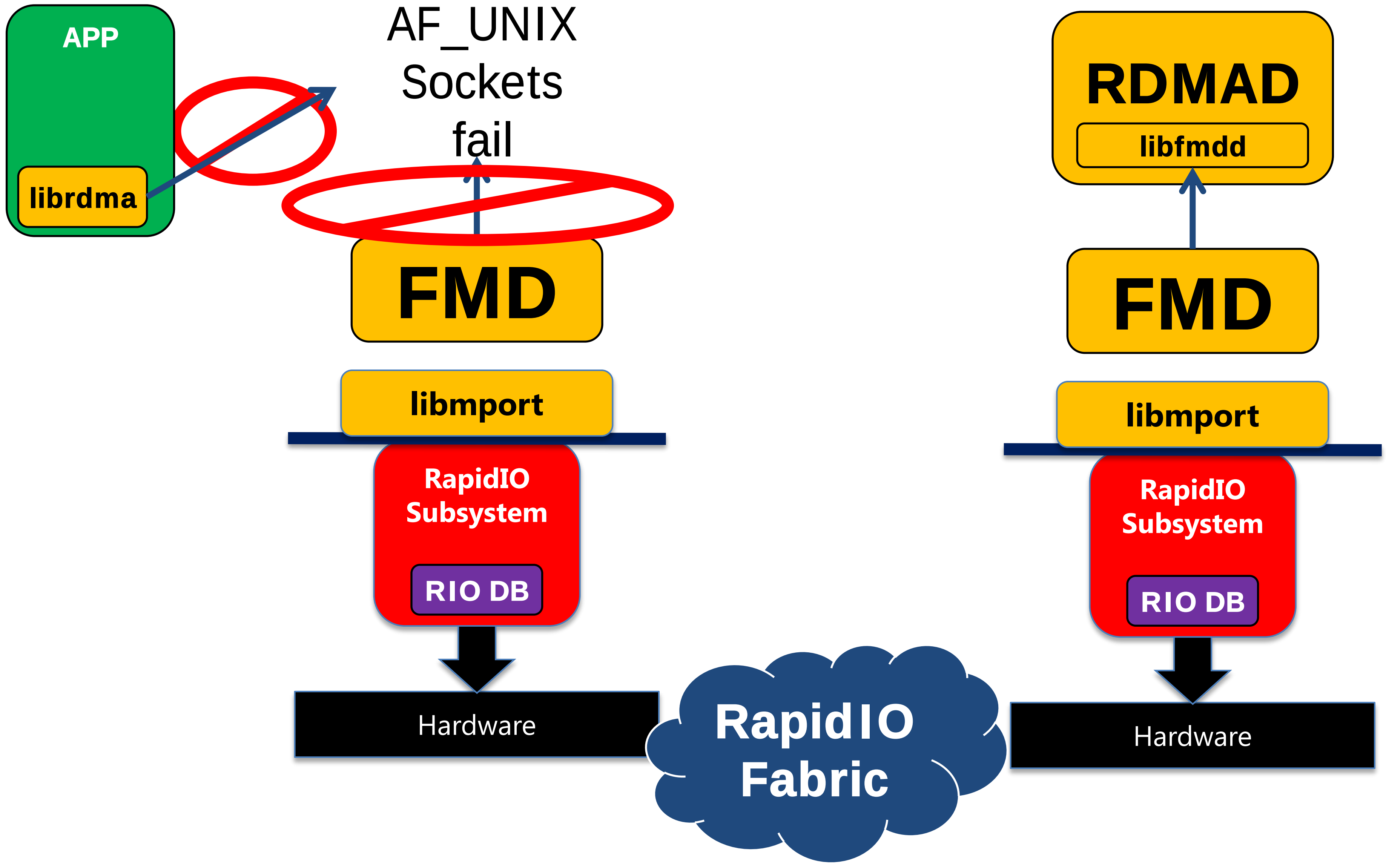
Master FMD can manage multiple different partitions, publish to different Fabric Directories

SECURE, SCALABLE, PARTIONABLE FABRIC MANAGEMENT

FMD Application Fault Tolerance

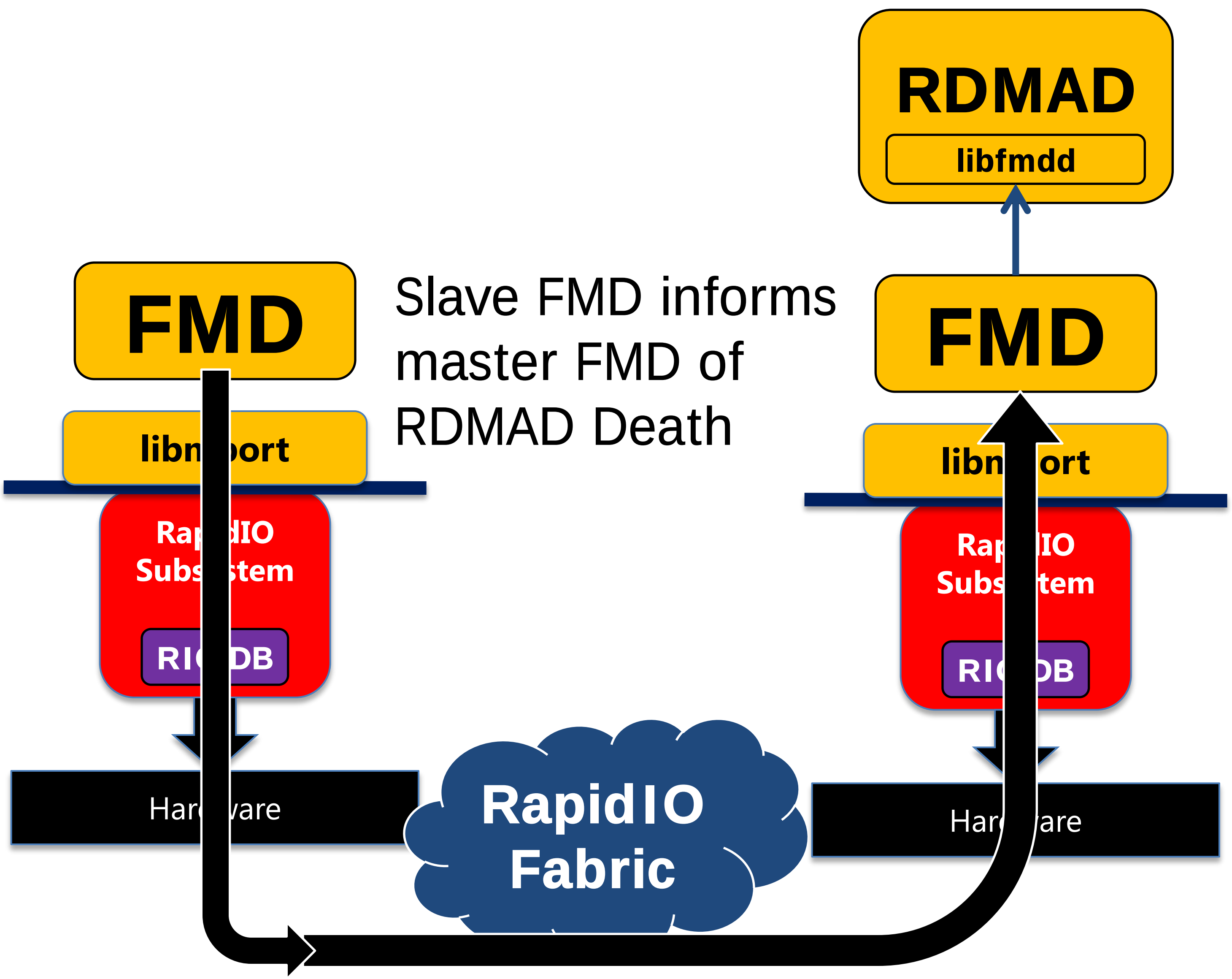


FMD Application Fault Tolerance

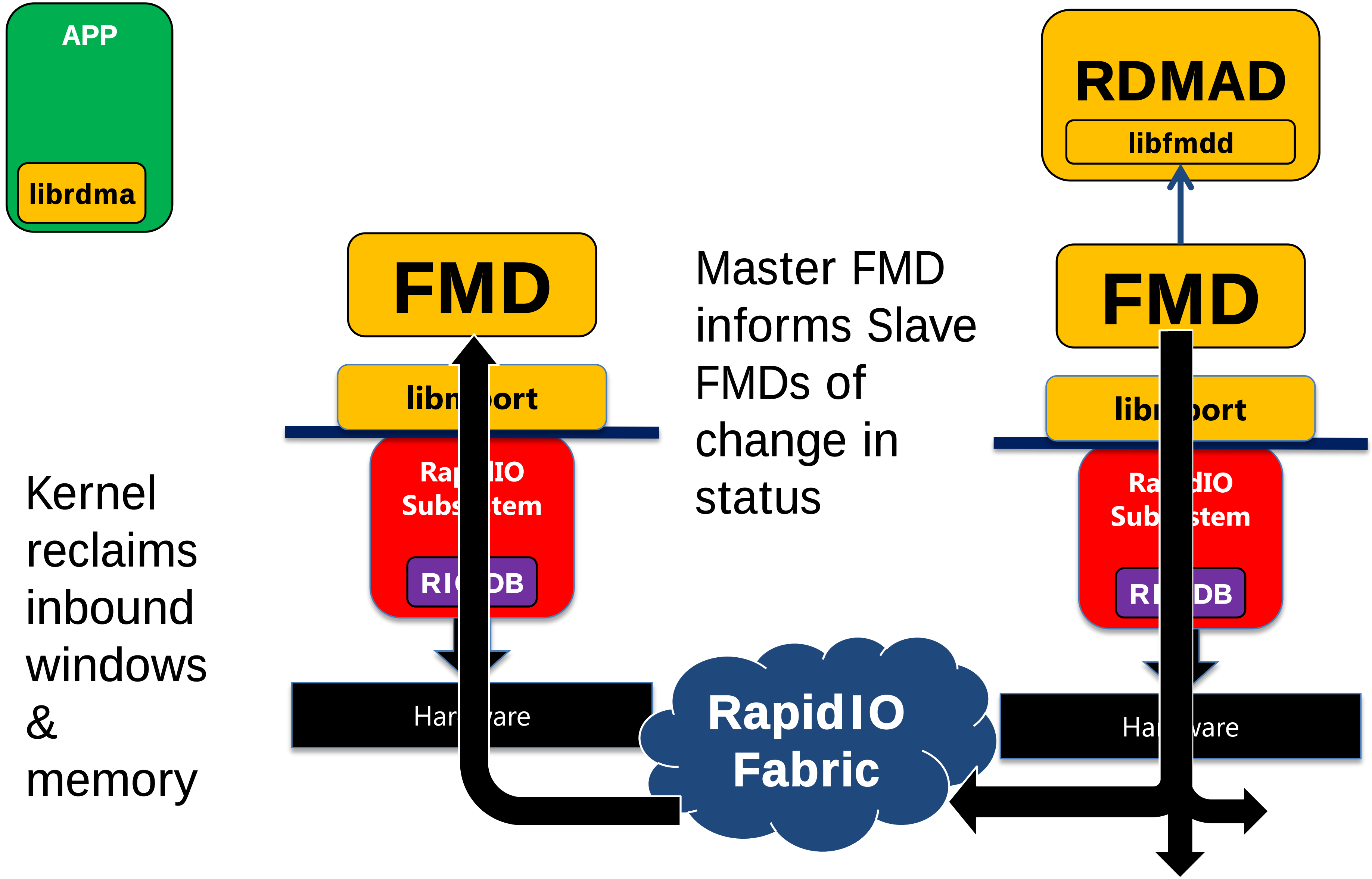


FMD Application Fault Tolerance

Library Halts All Accesses, Unmaps All Windows



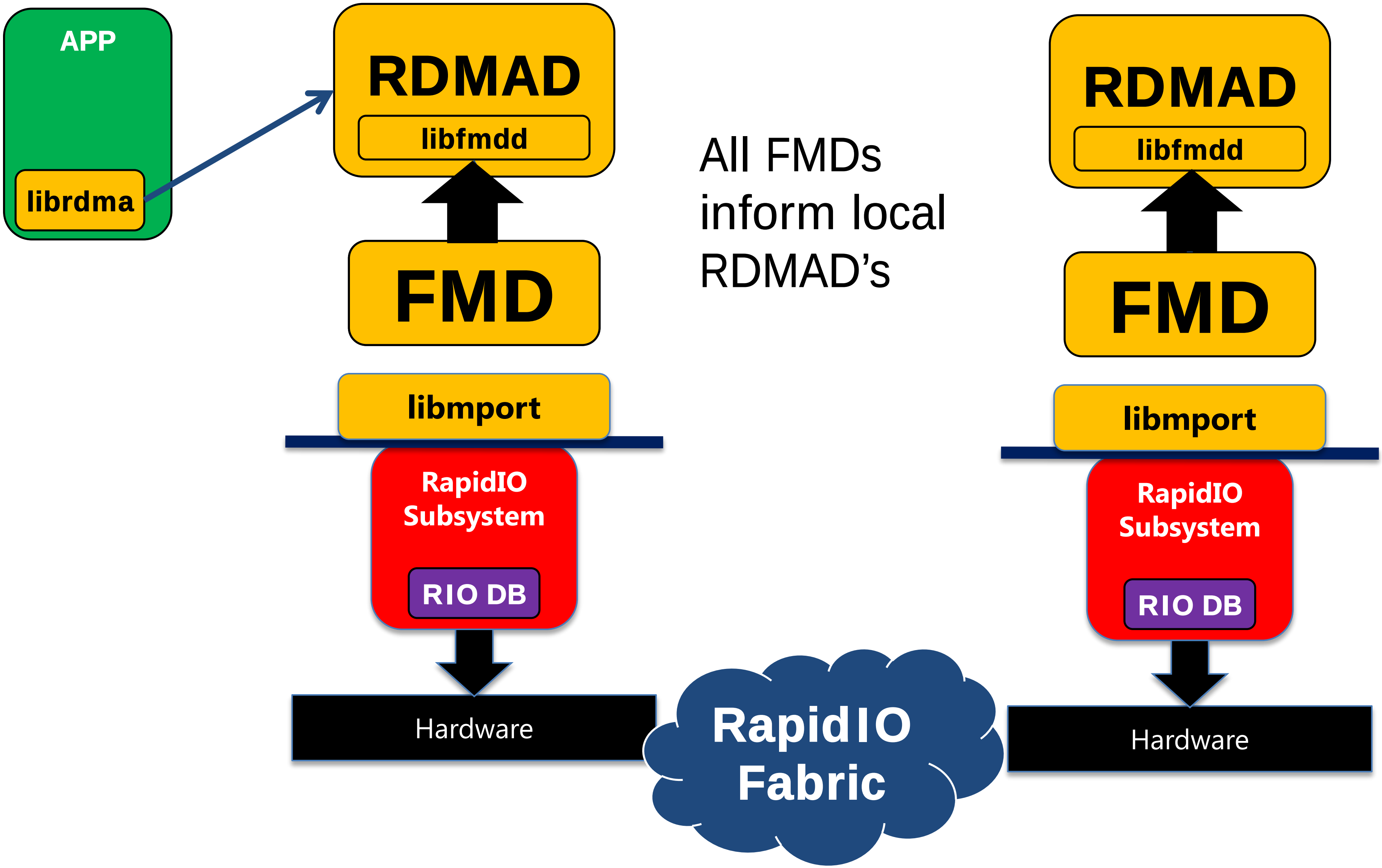
FMD Application Fault Tolerance



Kernel reclaims inbound windows & memory

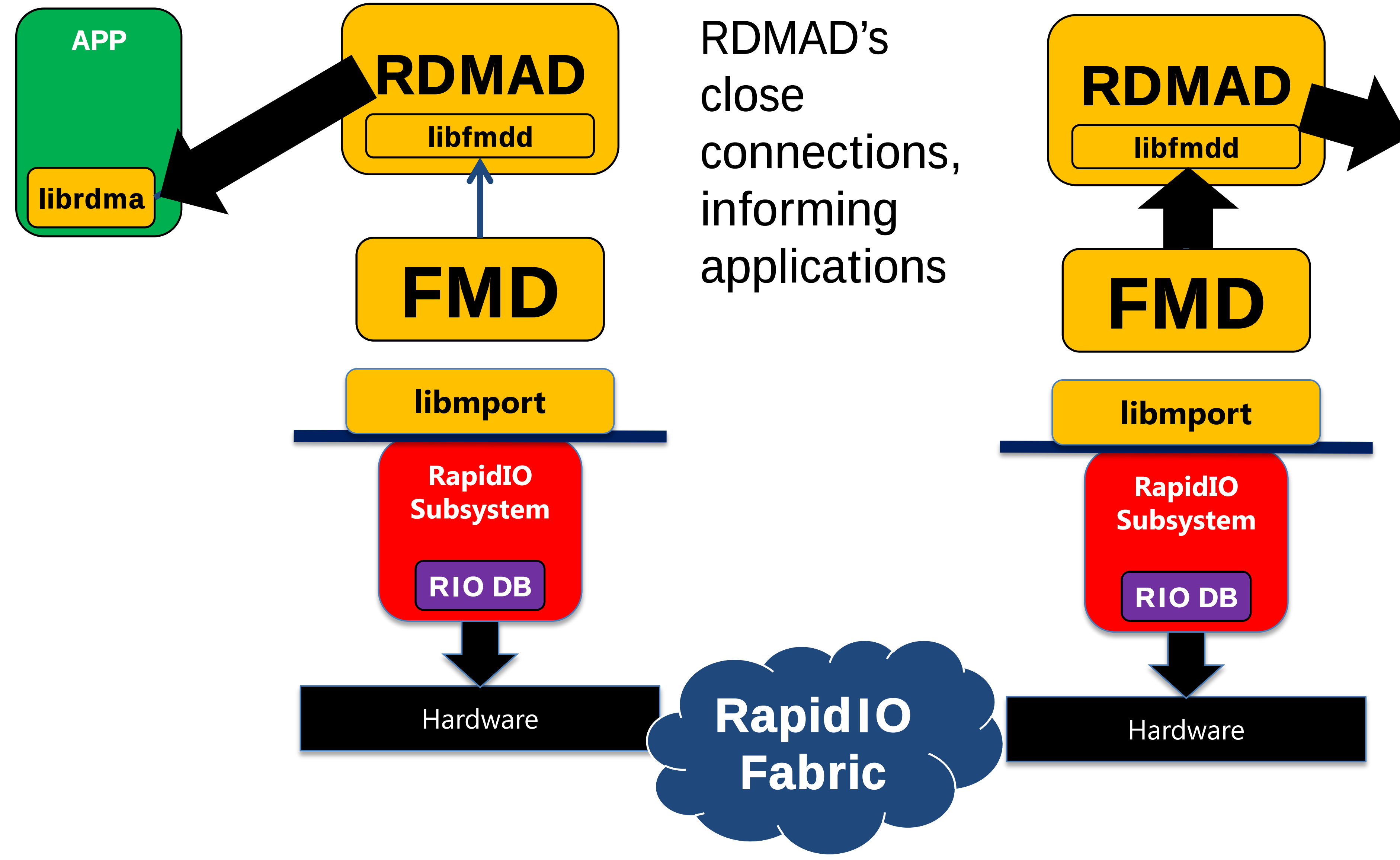
Master FMD informs Slave FMDs of change in status

FMD Application Fault Tolerance



All FMDs inform local RDMAD's

FMD Application Fault Tolerance



RDMAD's close connections, informing applications

RAPIDIO RDMA OVERVIEW

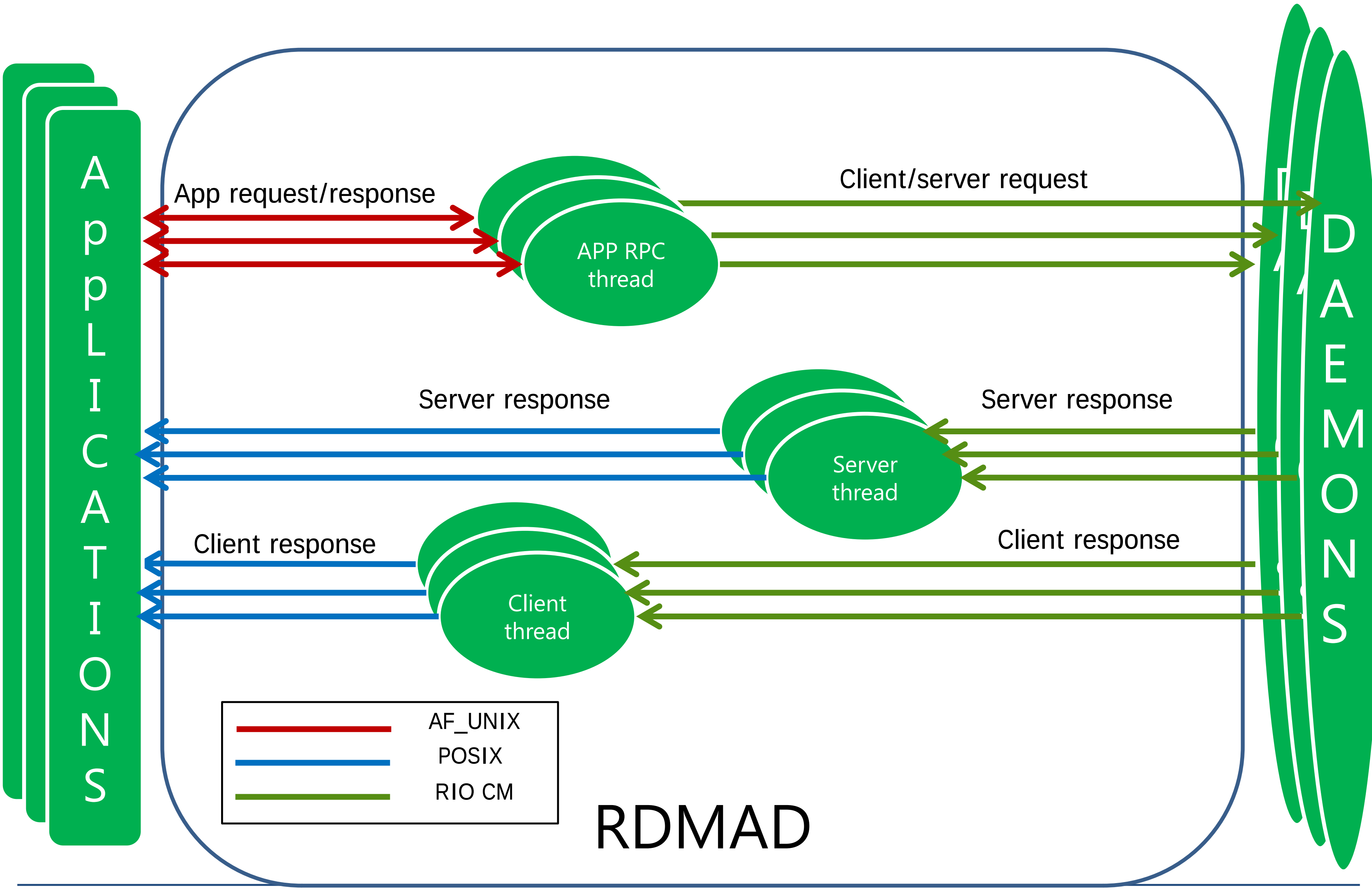
RDMA Interfaces: Memory Management

Routine	Description
rdma_create_mso_h	Owner: Create Memory Space Owner handle
rdma_open_mso_h	User: Open memory space owner handle
rdma_close_mso_h	User: Close memory space owner handle
rdma_destroy_mso_h	Owner: Destroy Memory Space Owner handle
rdma_create_ms_h	Owner: Create a Memory Space
rdma_open_ms_h	User: Open memory space
rdma_close_ms_h	User: Close memory space
rdma_destroy_ms_h	Owner: Destroy Memory Space
rdma_create_msub_h	User: Create a subspace handle for a Memory Space
rdma_destroy_msub_h	User: Destroy a subspace handle
rdma_mmap_msub	Memory map a subspace
rdma_munmap_msub	Destroy memory mapping of a subspace

RDMA Interfaces: Transfer

Routine	Description
rdma_conn_ms_h	Connect to a memory subspace on another device
rdma_accept_ms_h	Accept a connection request for a memory subspace
rdma_disc_ms_h	Destroy connection between remote and local subspaces
rdma_push_msub	Write date from local subspace to remote subspace
rdma_pull_msub	Read date from remote subspace to local subspace
rdma_push_buf	Write data from local buffer to remote subspace
rdma_pull_buf	Read date from remote subspace to local buffer
rdma_sync_chk_push_pull	Push and pull routines support DMA completions: • Blocking: Return when DMA is complete • Asynchronous: Continue, check for completion later • Fire and forget: Don't tell anyone rdma_sync_chk_push_pull is the "check for completion later" routine.

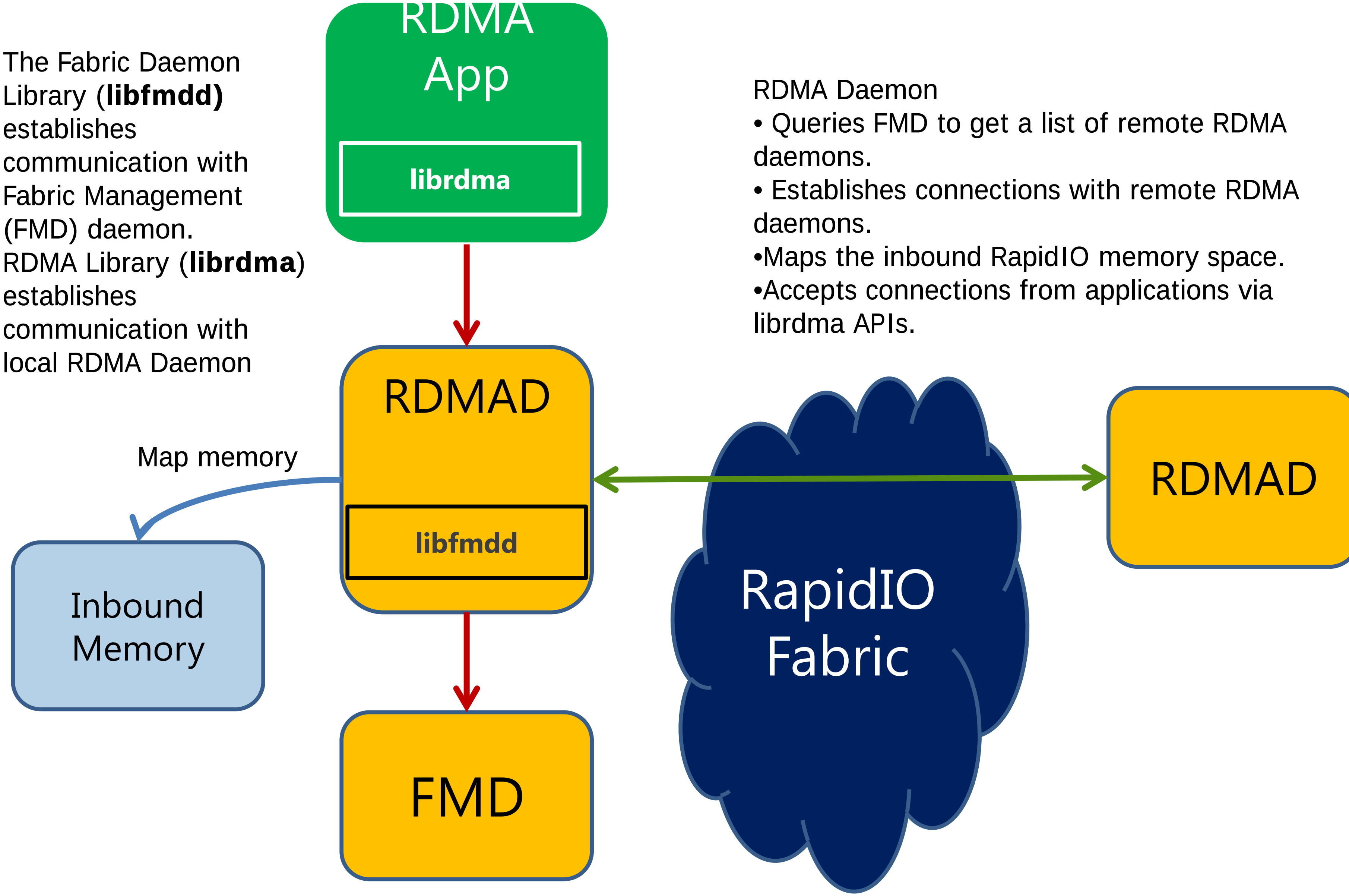
RDMAD Threads and Messaging



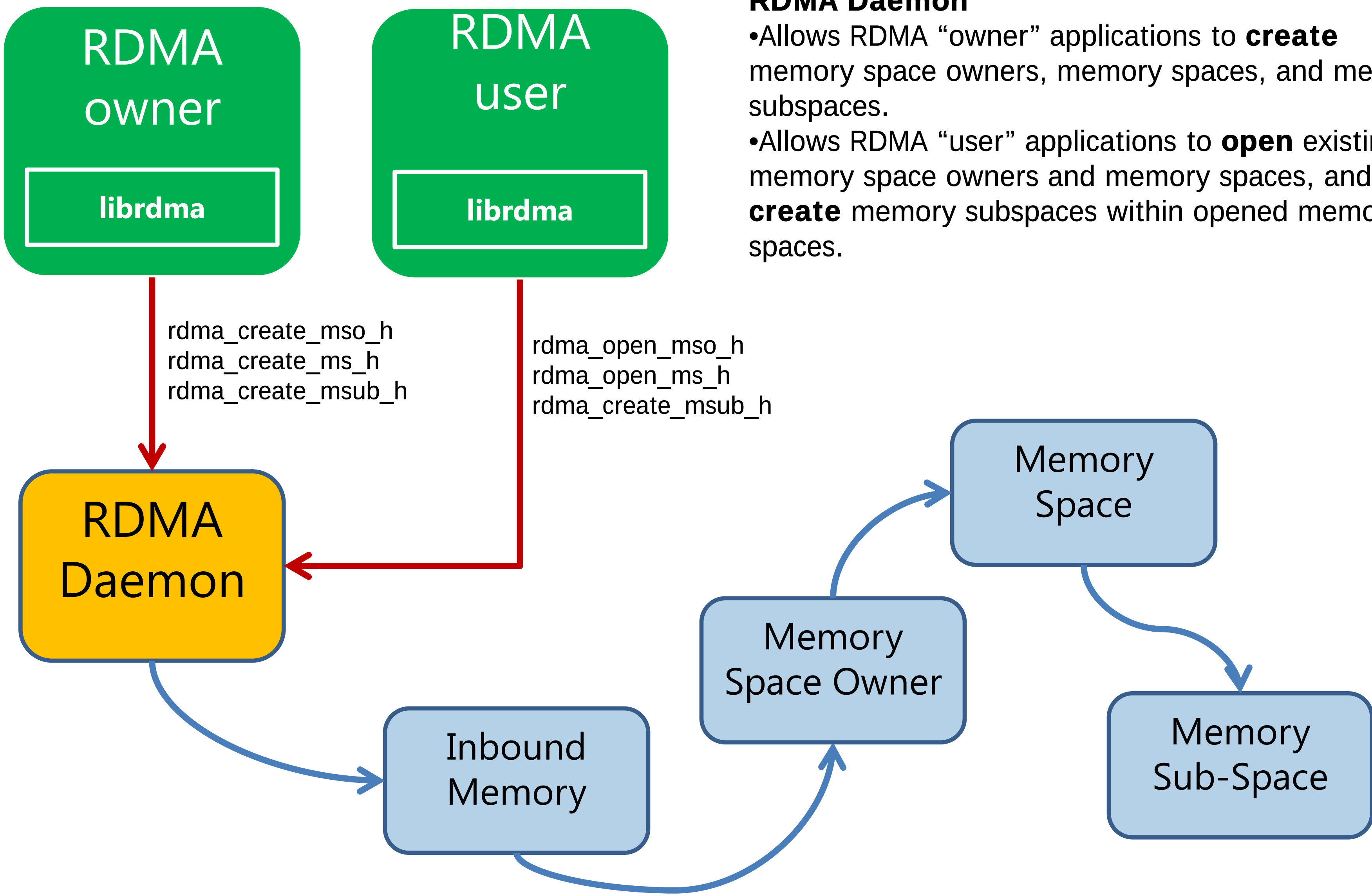
RDMA Daemon – Initialization

The Fabric Daemon Library (**libfmdd)** establishes communication with Fabric Management (FMD) daemon.
RDMA Library (**librdma**) establishes communication with local RDMA Daemon

RDMA Daemon
• Queries FMD to get a list of remote RDMA daemons.
• Establishes connections with remote RDMA daemons.
•Maps the inbound RapidIO memory space.
•Accepts connections from applications via librdma APIs.



RDMA Daemon – Memory Element Creation



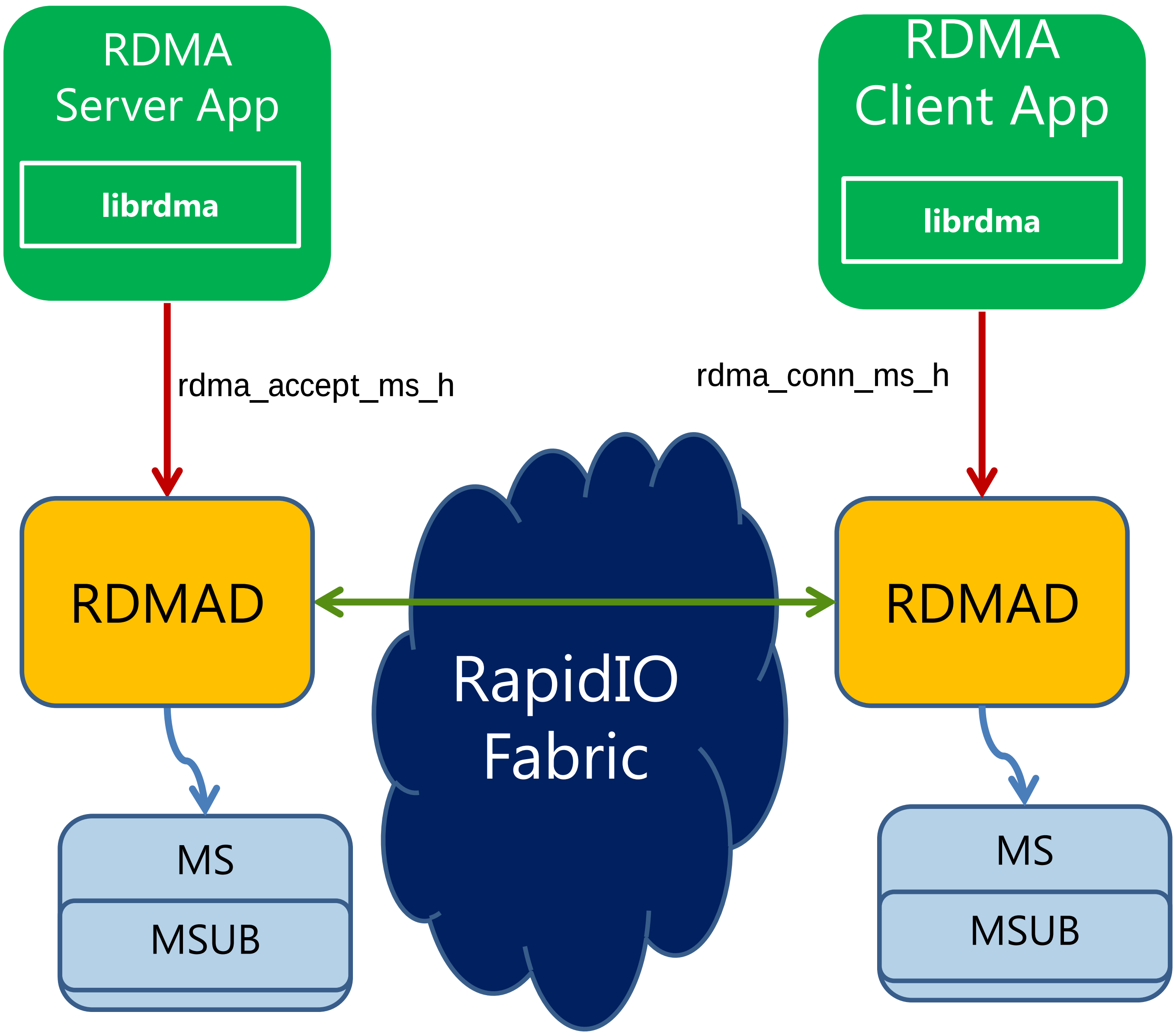
RDMA Daemon

- Allows RDMA “owner” applications to **create** memory space owners, memory spaces, and memory subspaces.
- Allows RDMA “user” applications to **open** existing memory space owners and memory spaces, and **create** memory subspaces within opened memory spaces.

RDMA Daemon – Connection Establishment

Server application calls rdma_accept_ms_h() on a memory space, and provides a memory subspace (msub) to client applications connecting to the memory space.
Client application calls rdma_conn_ms_h() and specifies the name of the remote memory space it wishes to connect to. It also provides an msub to the server.

rdma_accept_ms_h() is blocking.
rdma_conn_ms_h() has an optional timeout.

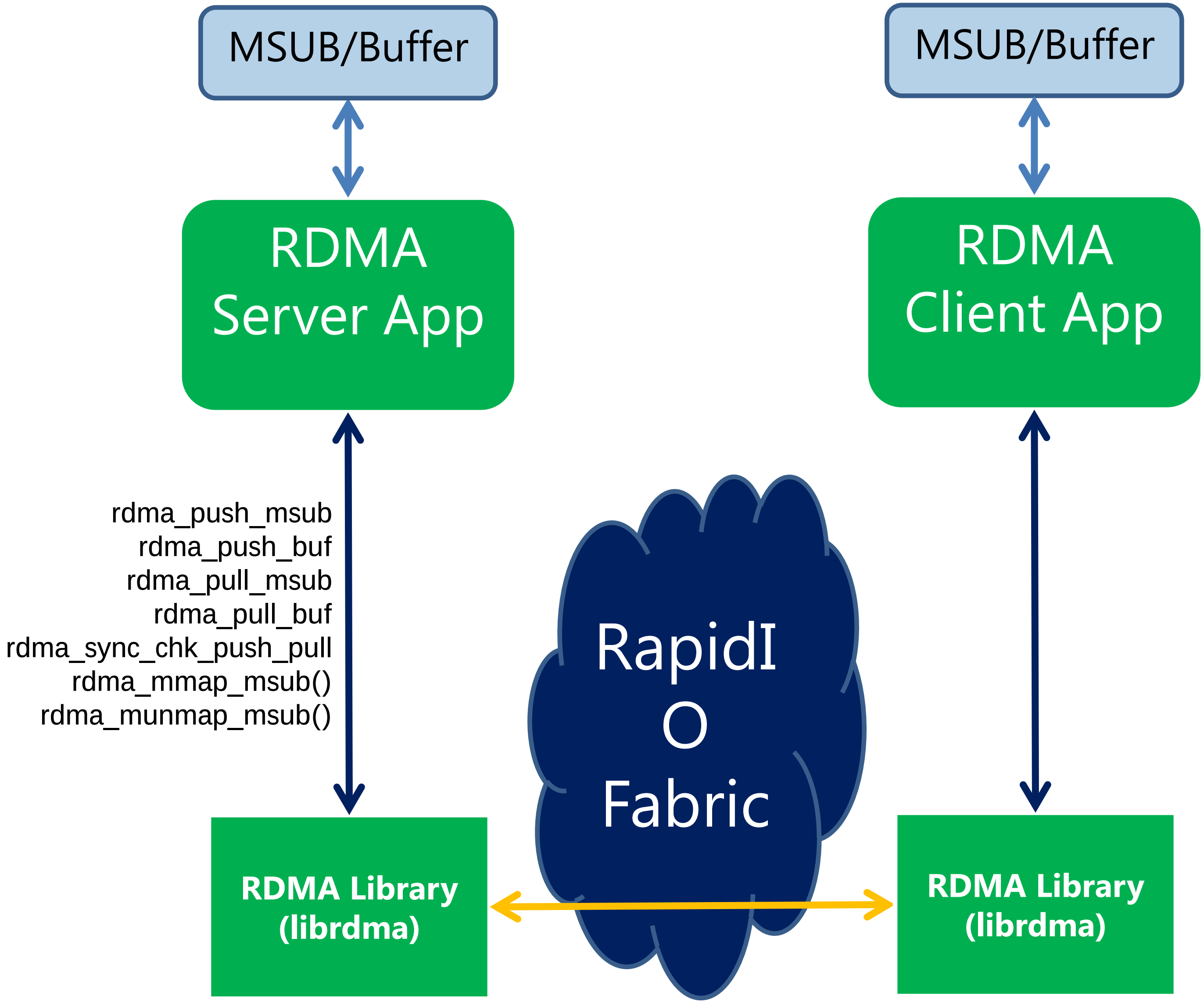


RDMA Daemon – DMA Transfers

Once connection is established between a server app and a client appl, DMA transfers can go directly between the two sides using only routines in the RDMA library.

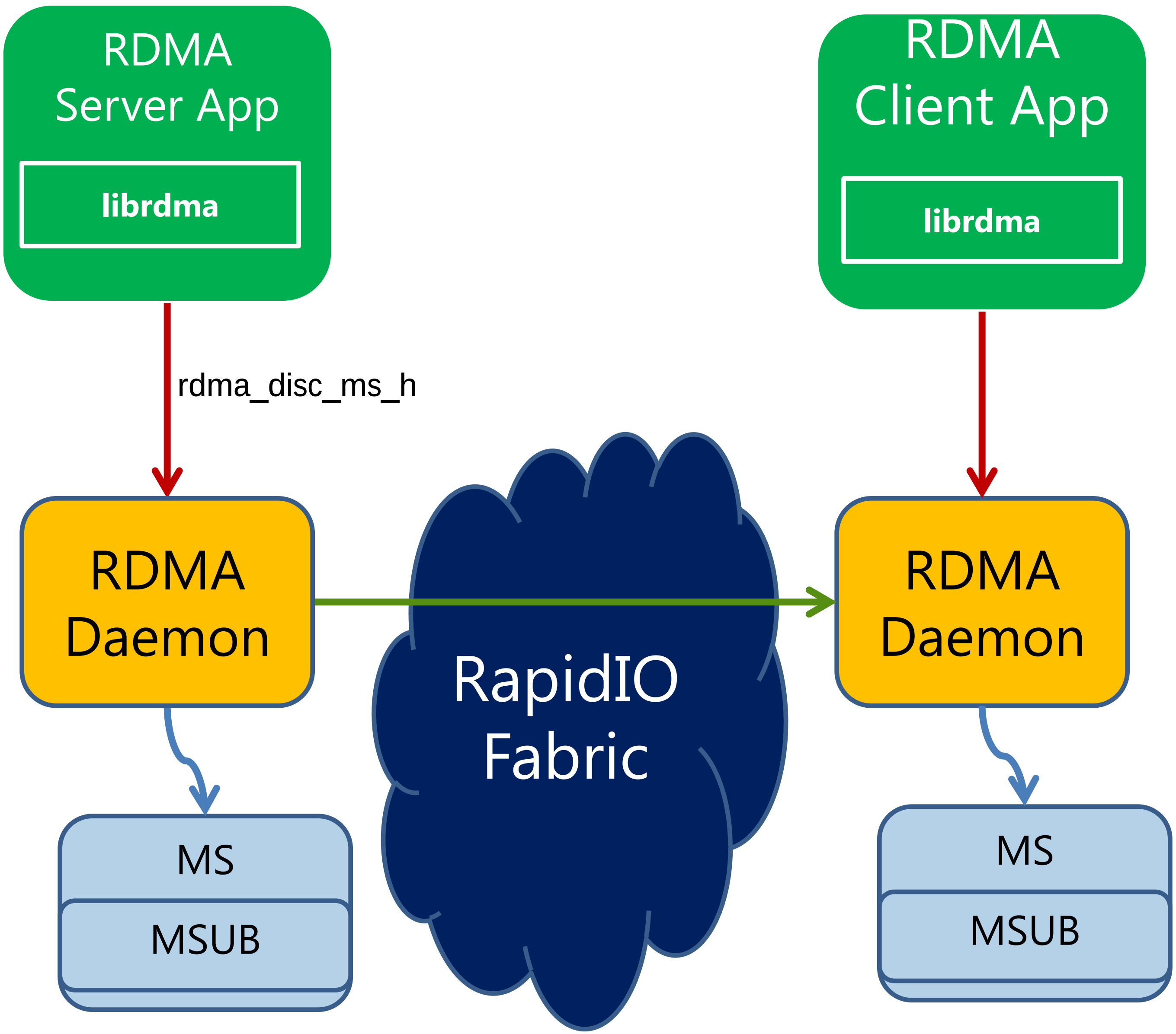
RDMA transfer routines provide push (write), and pull (read) capability in various modes (synchronous, fire-and-forget, and asynchronous).

Data can be exchanged using a memory subspace or a user buffer. For subspaces the space must first be mapped using rdma_map_msub().



RDMA Daemon – Connection Teardown

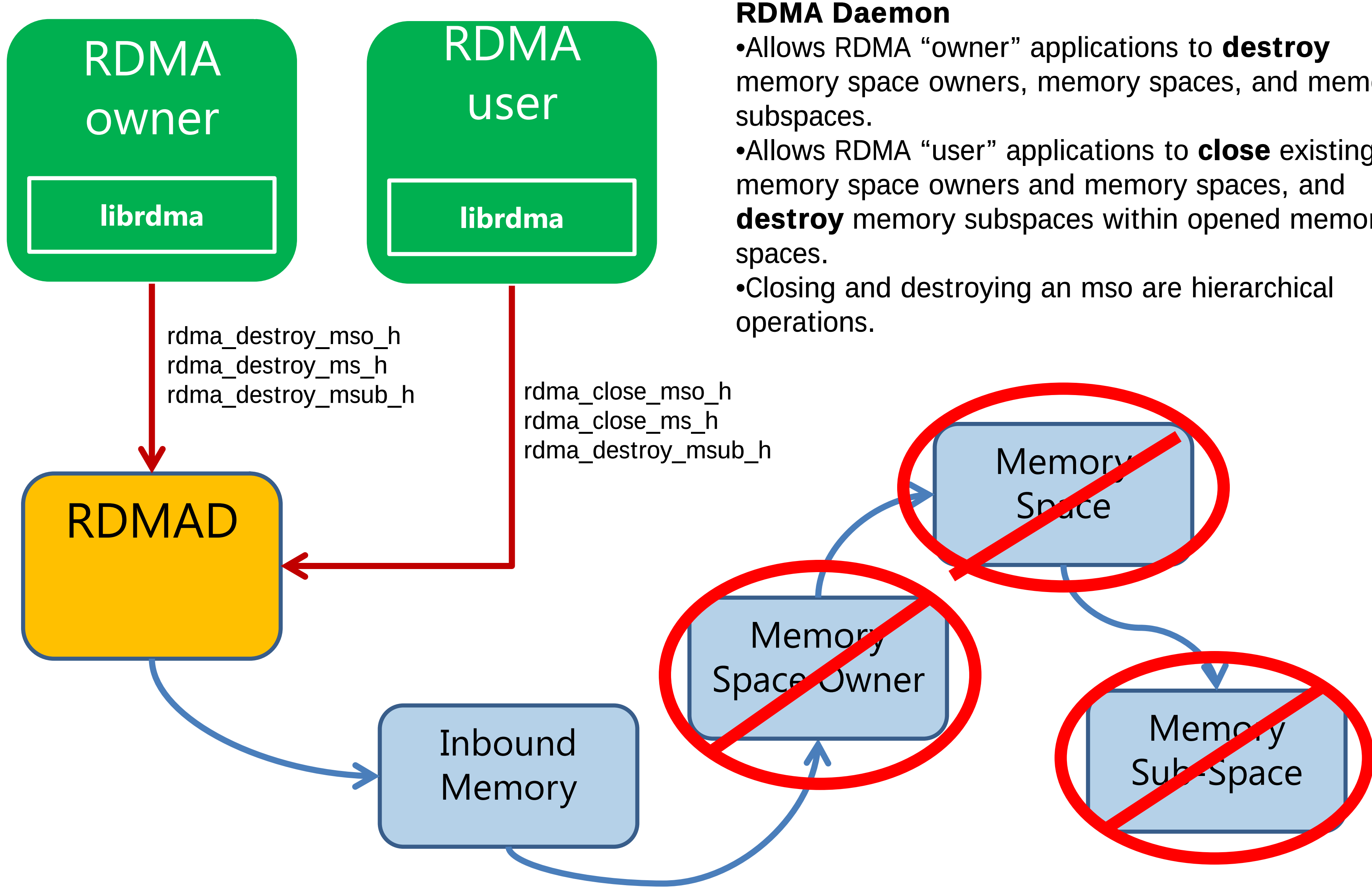
Client application calls rdma_disc_ms_h() and specifies the handle of the remote memory space it wishes to disconnect from.



RDMA Daemon – Memory Element Destruction

RDMA Daemon

- Allows RDMA “owner” applications to **destroy** memory space owners, memory spaces, and memory subspaces.
- Allows RDMA “user” applications to **close** existing memory space owners and memory spaces, and **destroy** memory subspaces within opened memory spaces.
- Closing and destroying an mso are hierarchical operations.

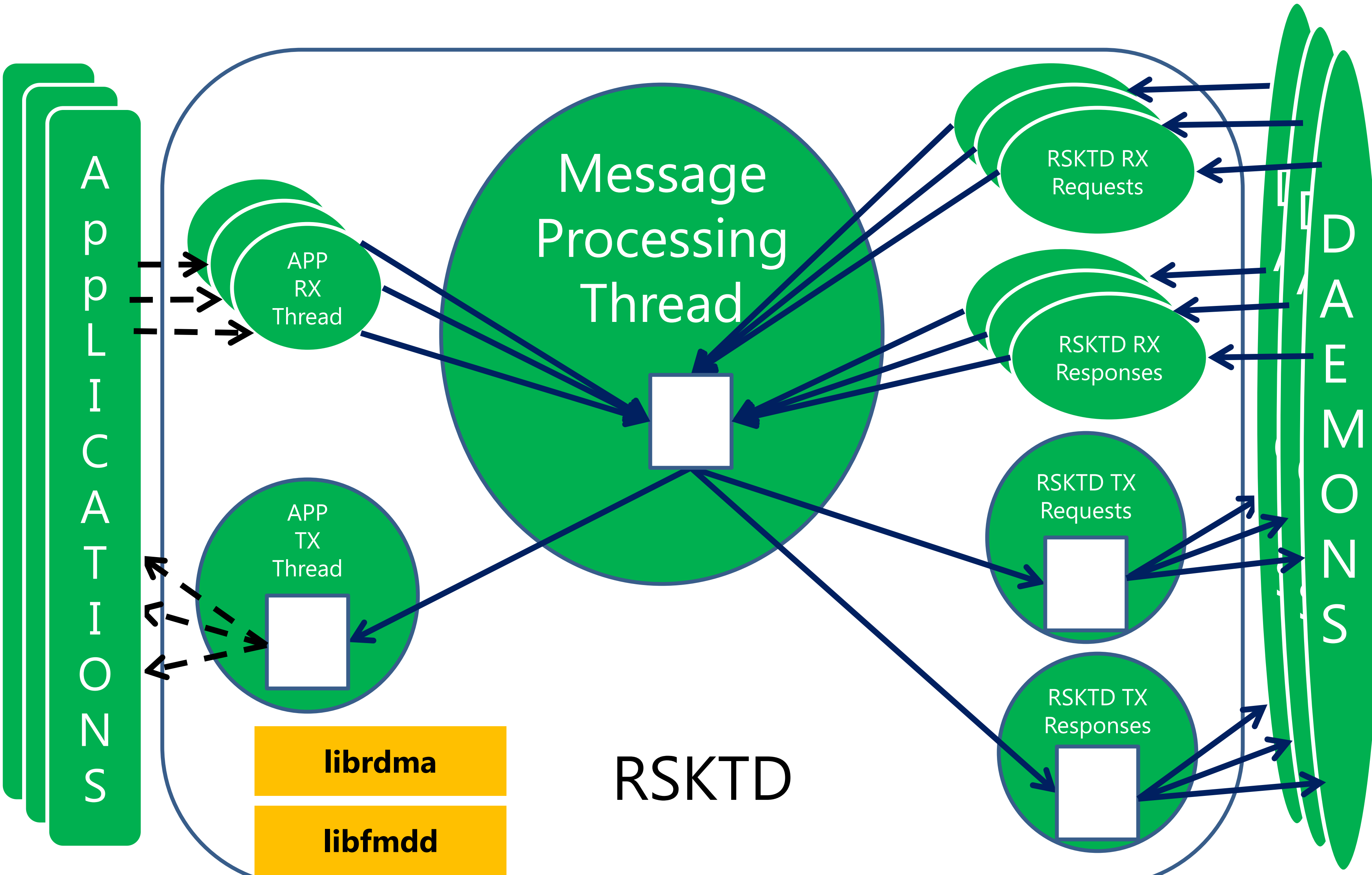


RDMA SOCKETS (RSKTS) OVERVIEW

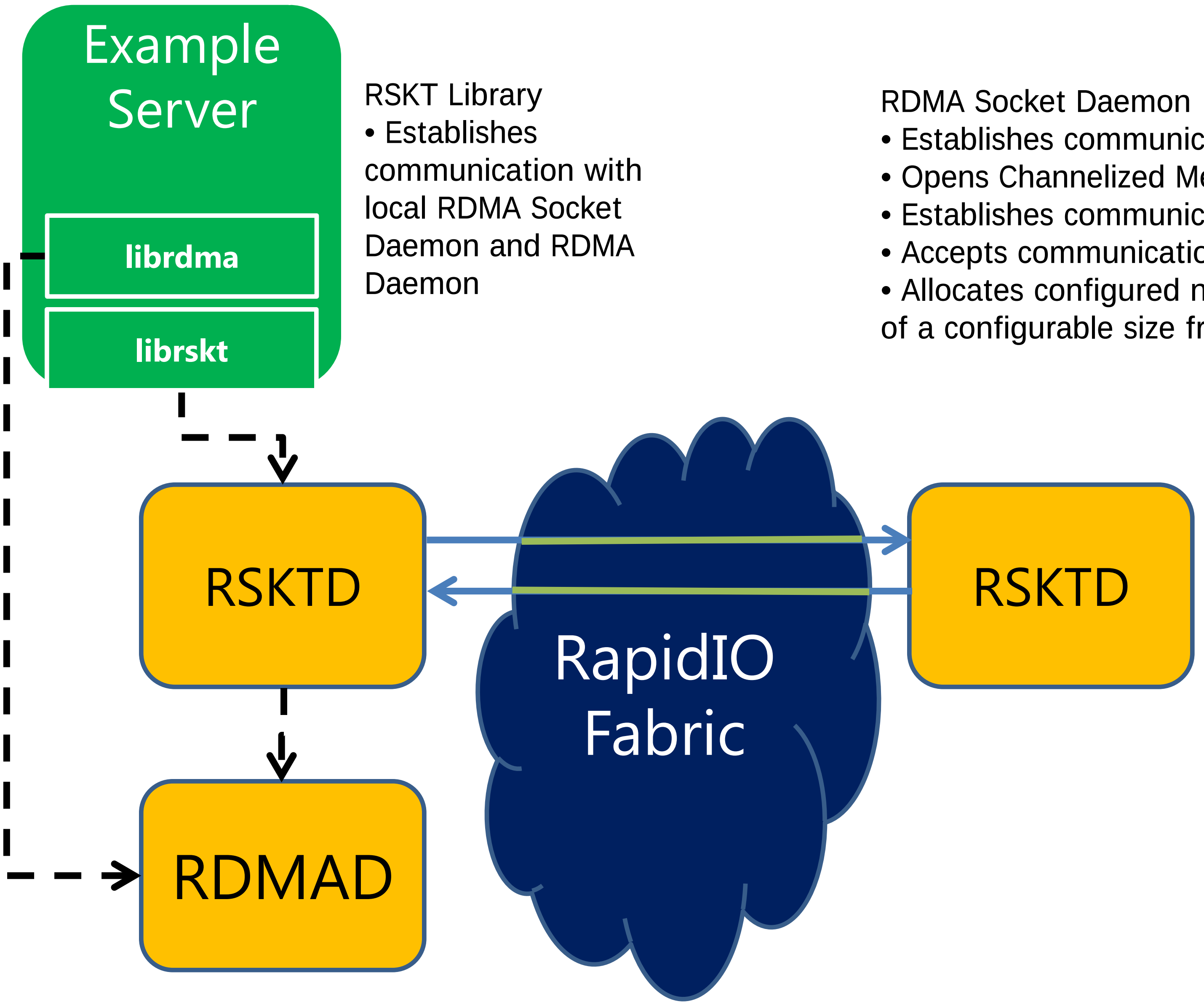
RDMA Sockets: Interfaces

Routine	Description
rskt_socket	Create socket data structure and opaque handle
rskt_bind	Bind socket to socket number
rskt_listen	Prepare to accept connect requests on this socket
rskt_accept	Accept a connect request to this socket
rskt_connect	Connect to an RDMA socket on another RapidIO node
rskt_write	Write data to socket connection
rskt_read	Read data from socket connection, stream of bytes
rskt_recv	Read next record from socket connection
rskt_flush	Flush all writes for socket connection
rskt_shutdown	Flush and then close the socket connection
rskt_close	Close the socket connection, may or may not transfer all data before closure.

RSKTD Threads and Messaging Queues



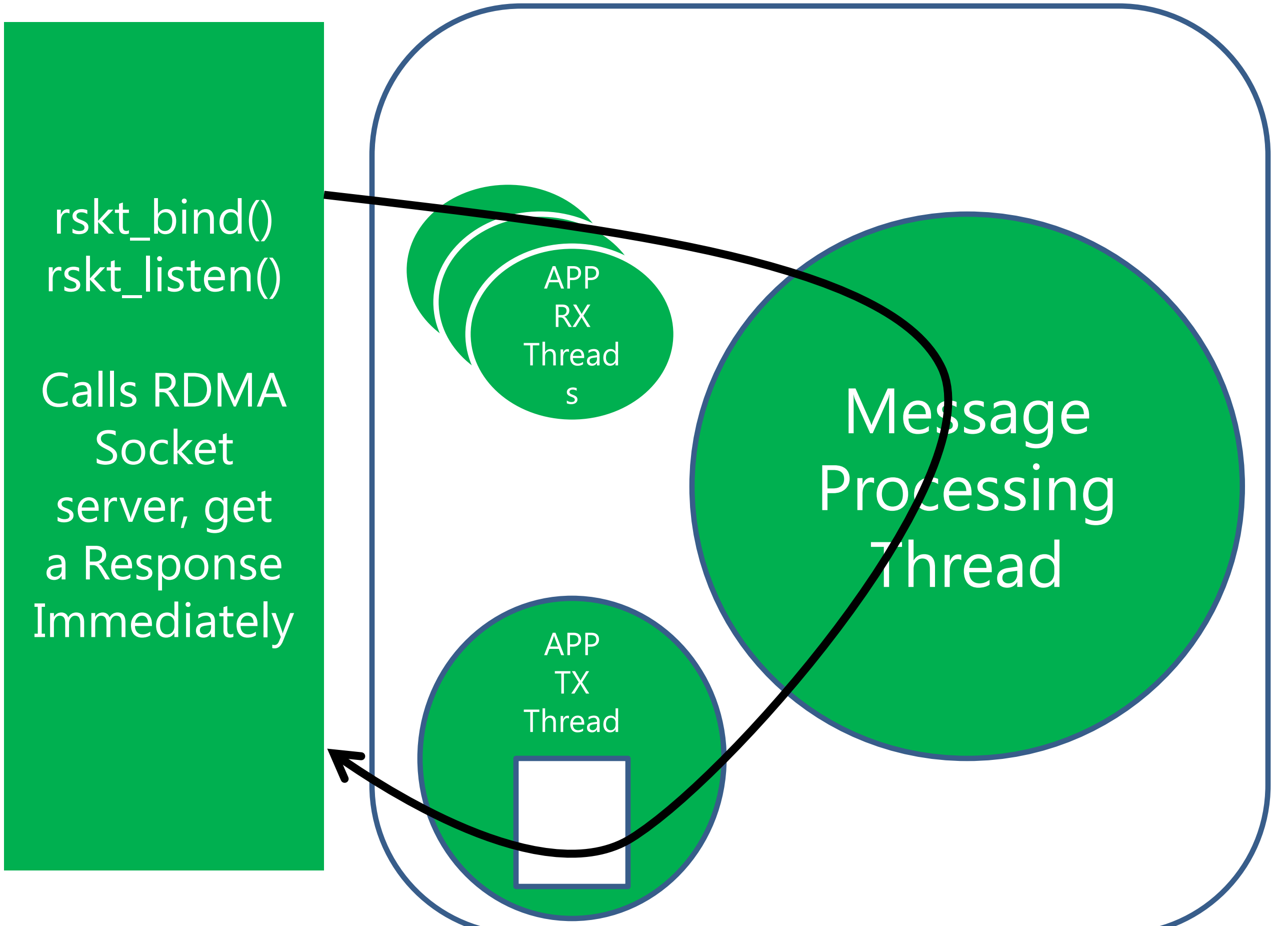
RDMA Socket Daemon – Initialization

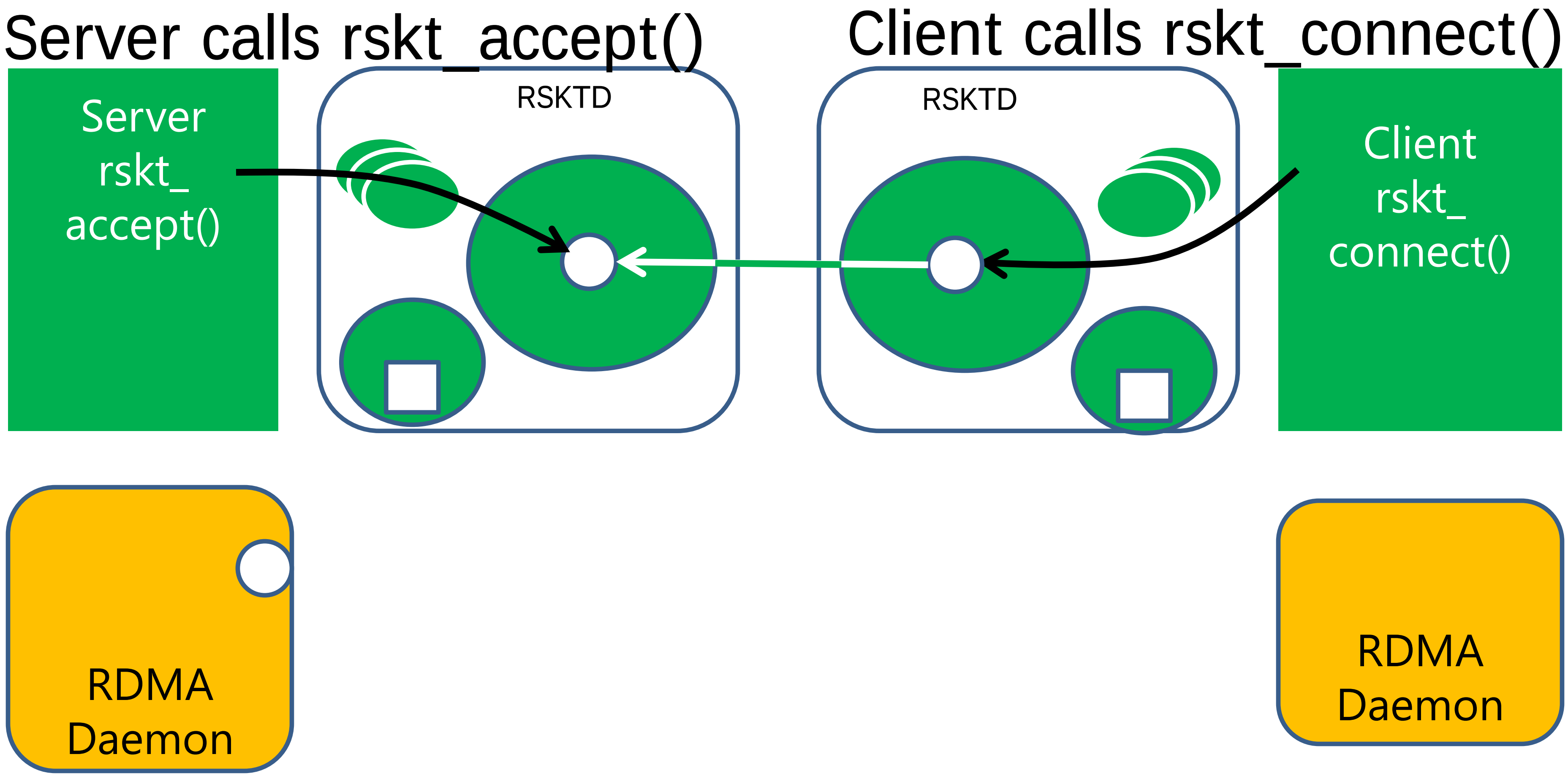


RSKT Library
• Establishes communication with local RDMA Socket Daemon and RDMA Daemon

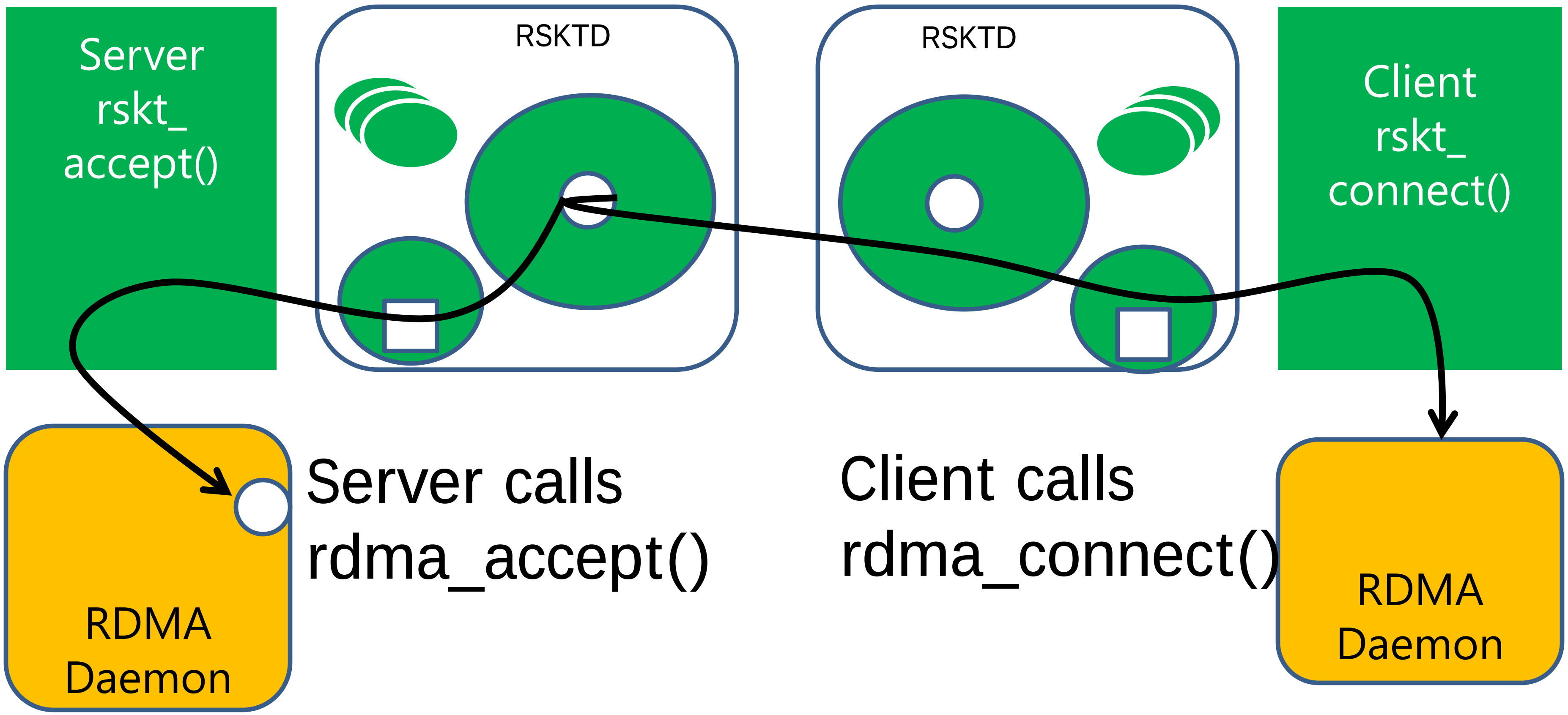
RDMA Socket Daemon
• Establishes communication with local RDMAD
• Opens Channelized Messaging socket
• Establishes communication with remote RSKTD
• Accepts communication from remote RSKTD
• Allocates configured number of memory spaces of a configurable size from the RDMAD

RDMA Daemon – Bind and Listen

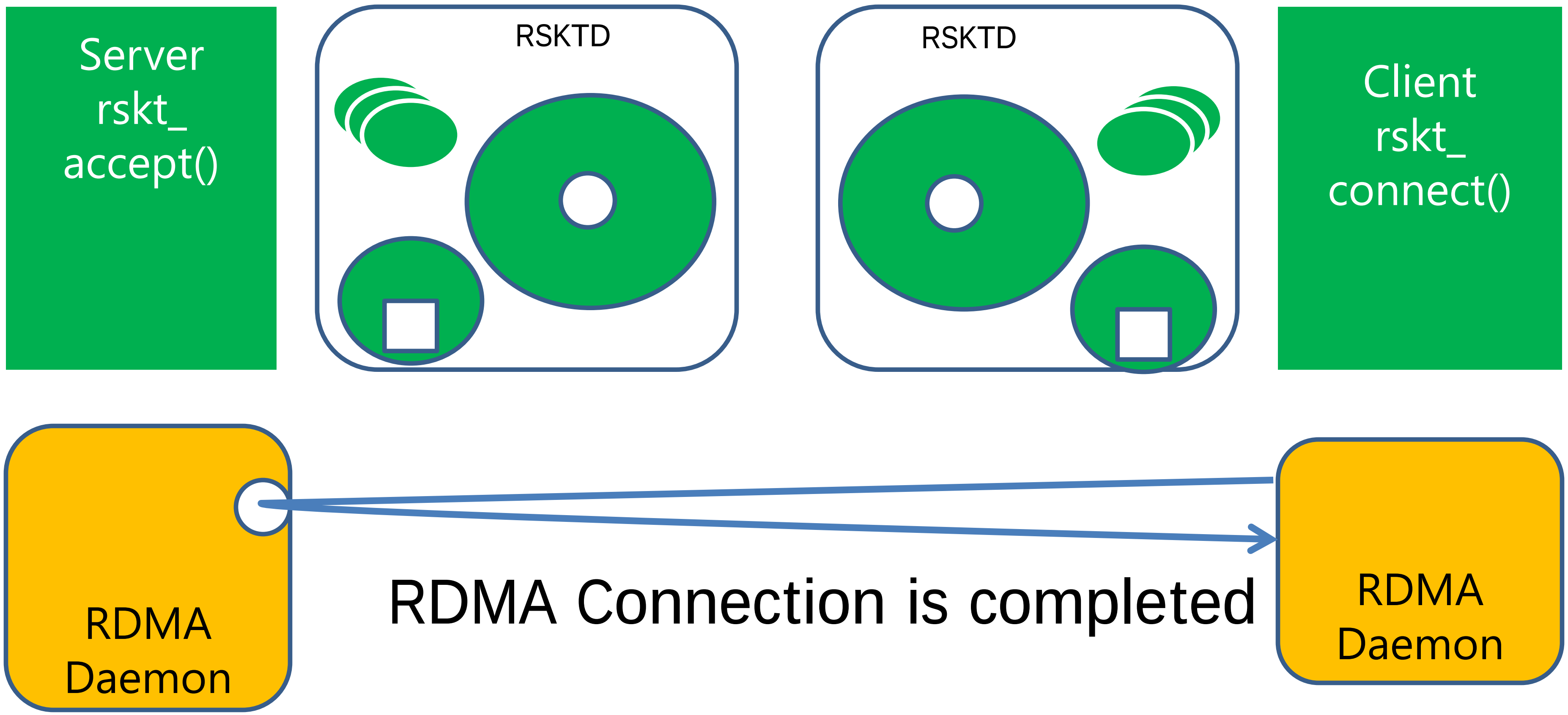



Server calls rskt_accept()
Client calls rskt_connect()
Server
rskt_
accept()
RSKTD
RSKTD
Client
rskt_
connect()
RDMA
Daemon
RDMA
Daemon

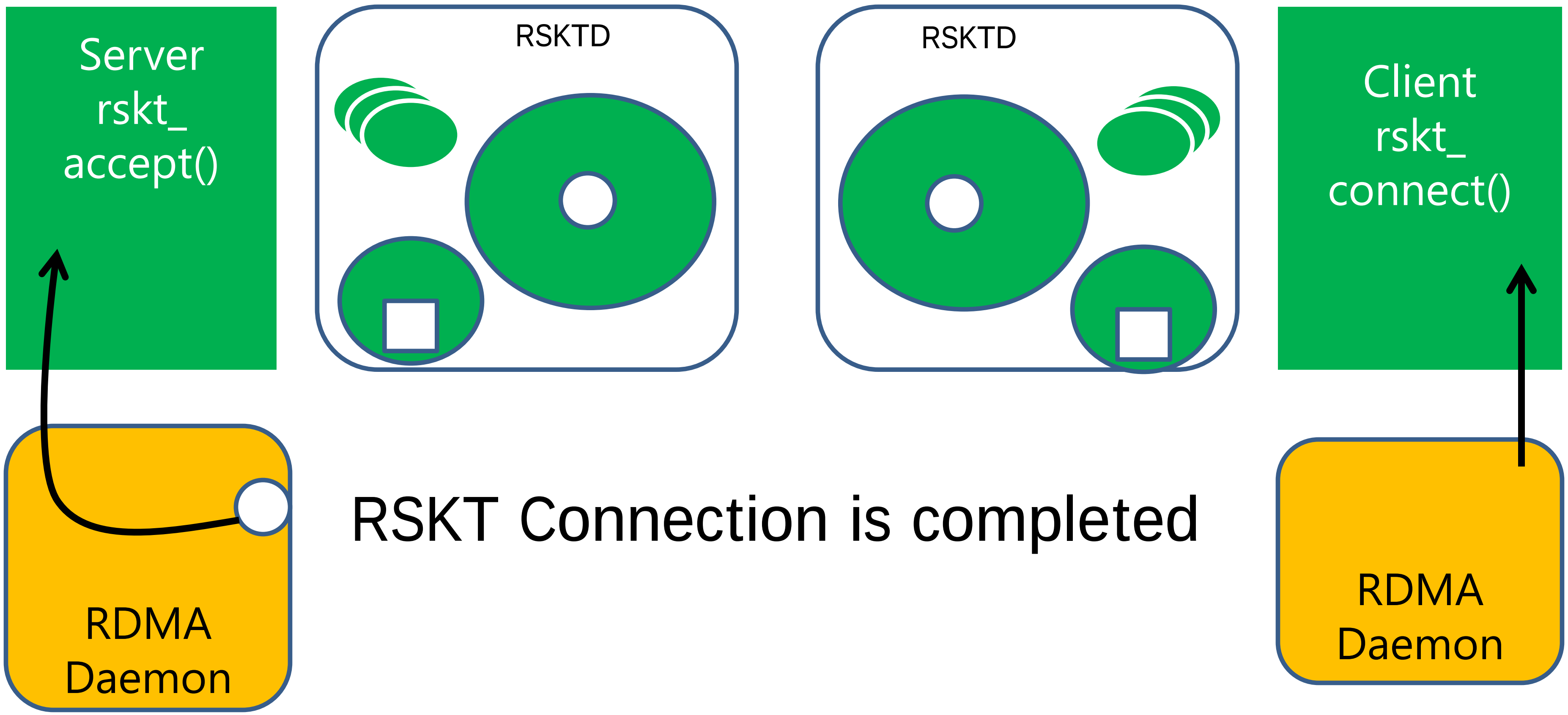
Connect and Accept



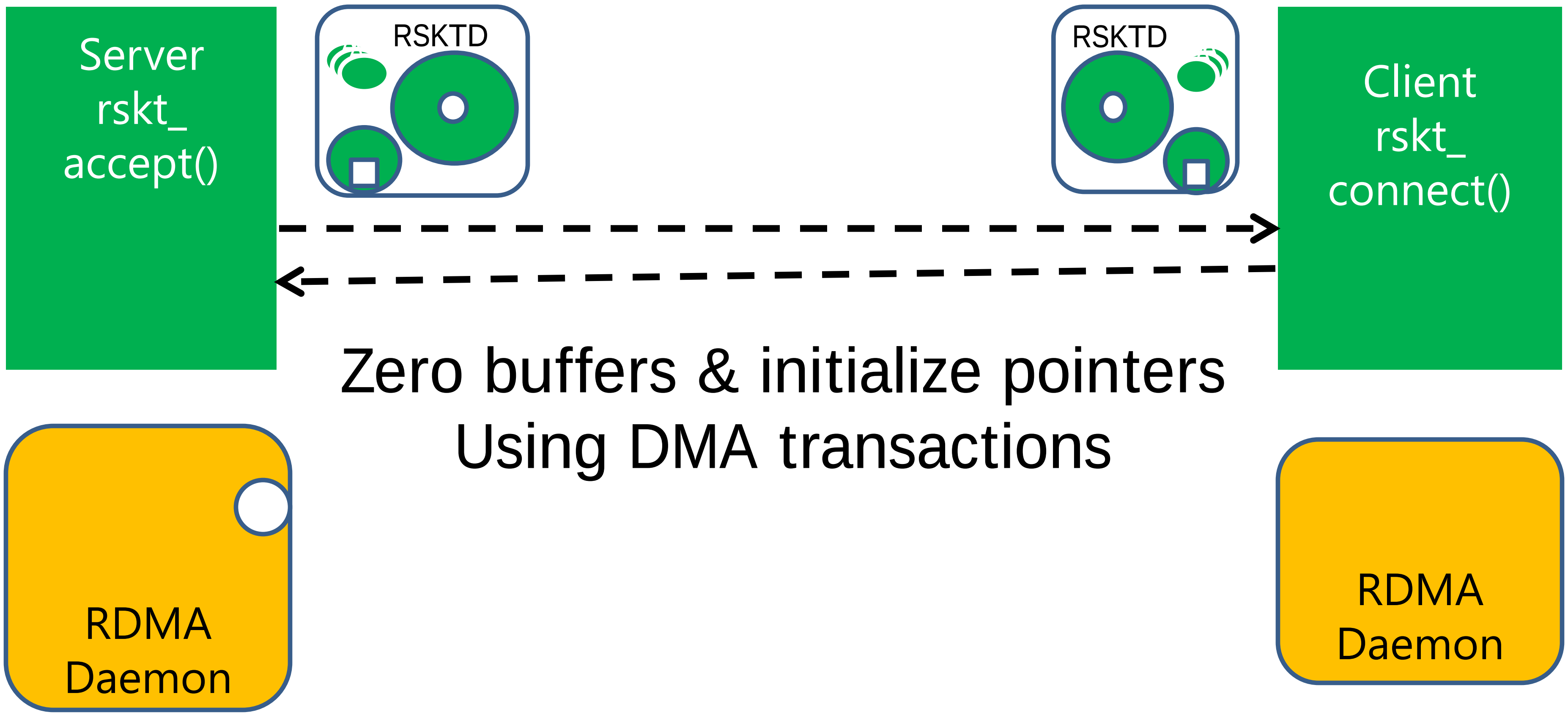
Connect and Accept



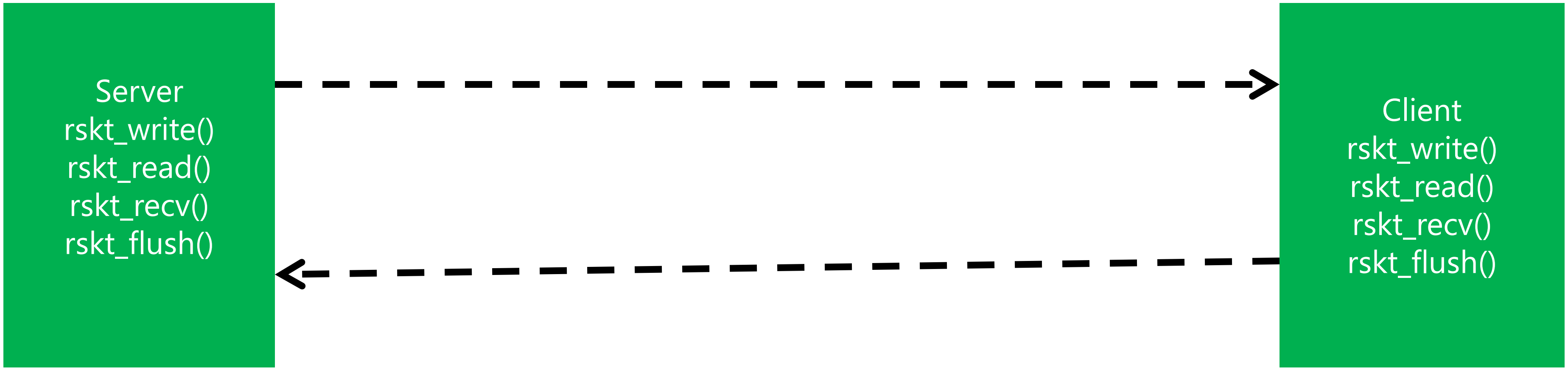
Connect and Accept



Connect and Accept



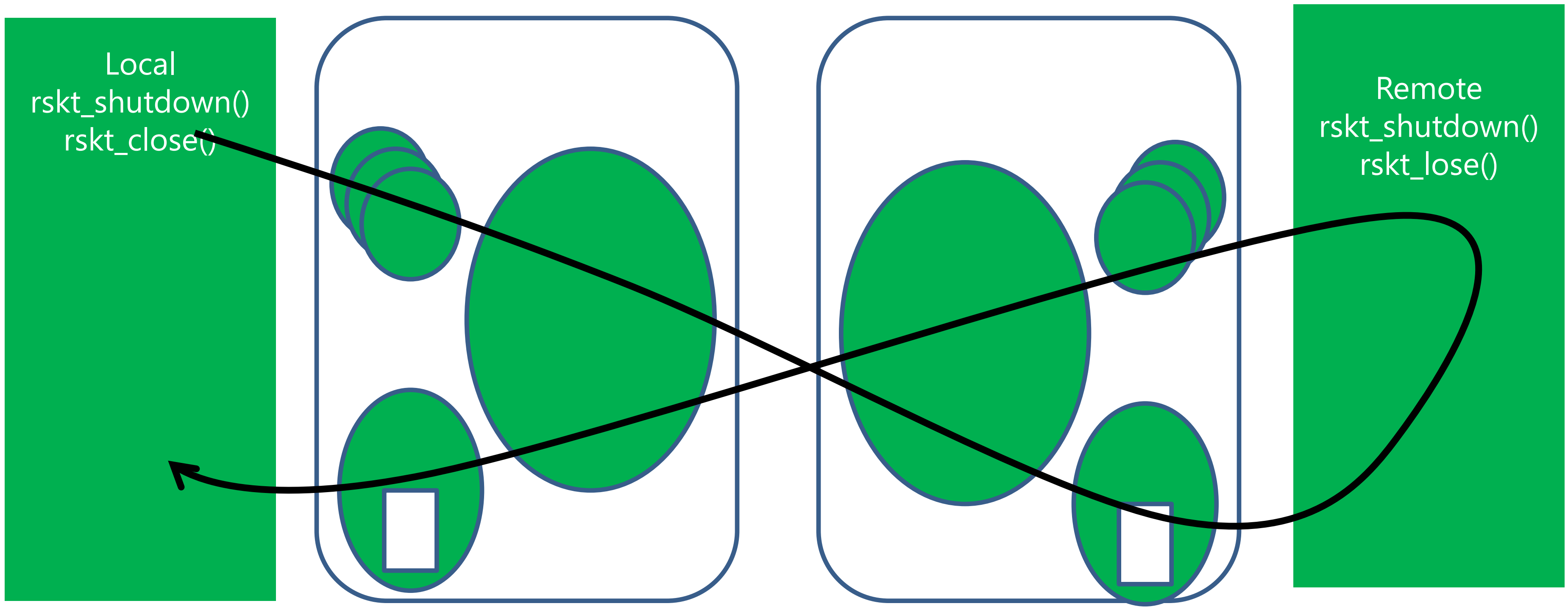
Write/Read/Recv/Flush



Write/Read/Recv/Flush all use RDMA library routines and DMA.

No messaging required.

Shutdown & Close



Shutdown and close are initiated

- By procedure call on a local client/server
- Automatically by failure of a DMA write/read/recv/flush request
- Automatically by closure of a POSIX socket with the local client/server
- Automatically by closure of the rio_cm_cdev connection with a remote RSKTD
- By request on a remote client/server

RAPIDIO SOFTWARE STACK DEBUG

Logging Library

Each API routine uses the “errno” global variable to report standard LINUX error codes on a failure.

In addition, each process and library reports status using the rapidio_sw/common/include/liblog.h

All log files are found in /var/log/rdma
- fmd.log
- rdmad.log
- rsktd.log

Logs can be displayed on each daemon using the “dlog” command.

Control of Logging

The level of detail for all logs is controlled using the “levelog” command, available from the CLI for each daemon
- To access the CLI prompt for each daemon, use “screen –r <app>”
- For example, to connect to the RDMAD for debug, use “screen –r rdmad”

Additional debug commands are available from the CLI for process debug

The first step in reporting a problem is examining/transferring the log files found in /var/log/rdma/...
- fmd.log-<date>
- rdmad.log-<date>
- librdma.log-<date>
- rsktd_log.txt-<date>

RAPIDIO FILE TRANSFER UTILITY CHARTS SEPTEMBER 2015

File Transfer Overview

Server

- Acts as a target for “send file” requests
- Manages externally available memory
- Manages multiple large (~2 MB)memory segments as “ping pong” buffers

Library (libfxfr)

- exposes “send_file” routine to send a file to the server

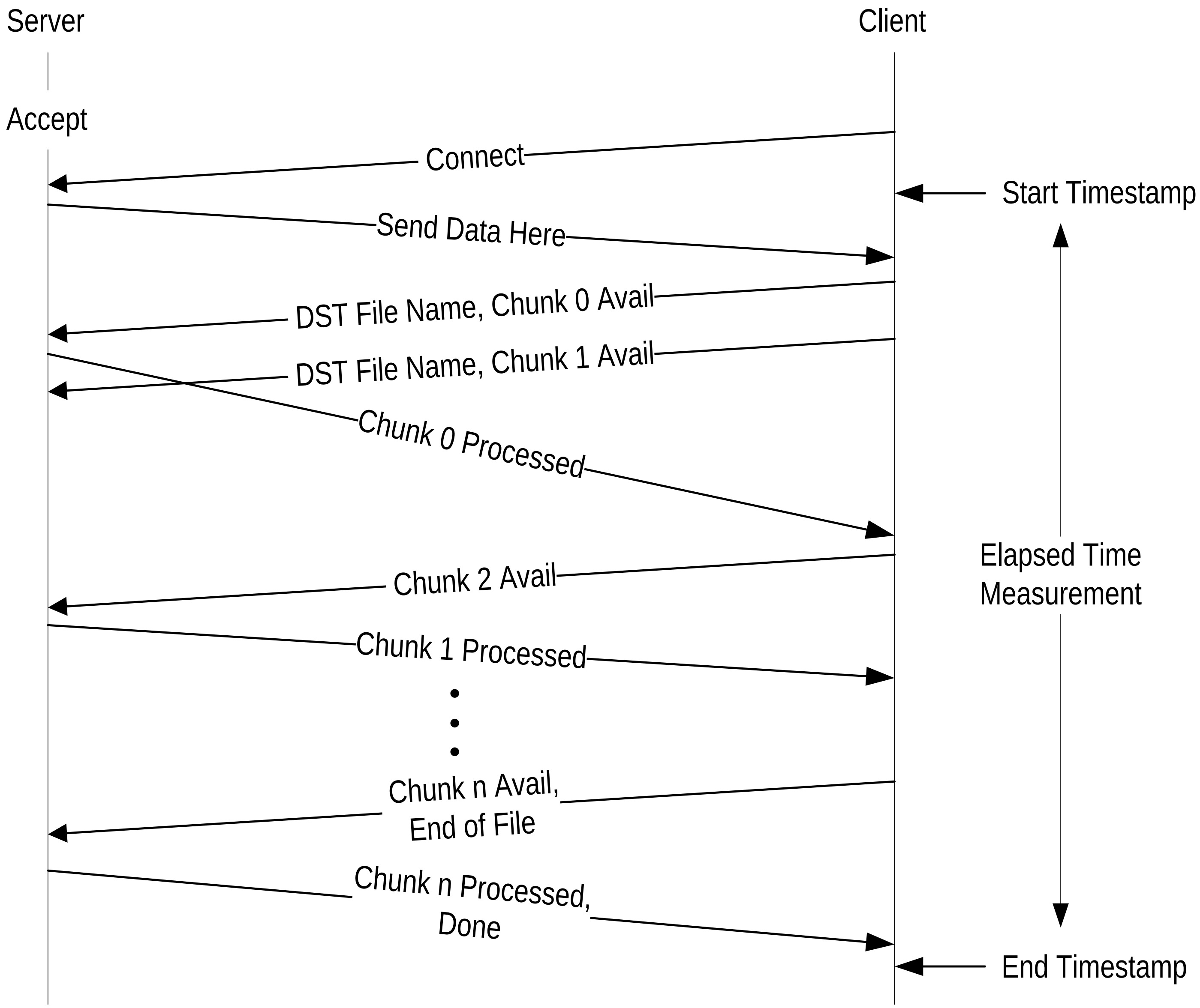
Performance

- Much better for large (multi megabyte) files, rather than small files
- RapidIO transfers can occur at line rate (1700 MBps), even with Kernel Driver
- Limited by the speed of the media
    - SAS can’t handle 1700 MBps...

File Transfer Implementation

File	Description	Server	libfxfr
server.c	FXFR Server Main Loop	Yes	
libfxfr.h	Client interfaces for libfxfr		Yes
libfxfr_private.h	Server interfaces for libfxfr	Yes	Yes
fxfr_msg.h	Channelized messaging formats for libfxfr	Yes	Yes
fxfr_rx.c	Implementation of file receive	Yes	
fxfr_tx.c	Implementation of send_file		Yes

File Transfer Message Sequence Chart



GOODPUT TOOL OVERVIEW

Goodput/Ugoodput Capabilities

Measurements

- Goodput
- Latency

Transaction types

- Direct I/O
- DMA
  - Sync, async, fire-and-forget
- Messaging

Drivers

- libmport
- "Demo" user mode driver

Goodput/Ugoodput Structure

Single process, multiple threads
- Command line interpreter (CLI) thread
-12 worker threads

CLI functions
- start/stop/configuration of worker threads
- dispatch tasks to workers
- measurement display

Bash Scripts
- generate CLI scripts for automated measurements
- summarize logged results into simple tabular format

Goodput/Ugoodput Instructions

To compile goodput/ugoodput:
Type “make all” in rapidio_sw/utils/goodput directory

Goodput documentation
-Type “doxygen doxyconfig” in rapidio_sw/utils/goodput
- Open rapidio_sw/utils/goodput/html/index.html file with browser OR
- Refer to goodput/inc/goodput_intro.h