IB Gold Distribution Stack
User's Manual

Rev 1.70

IB Gold Distribution Stack User’s Manual

Document Number: 2142UM

Mellanox Technologies, Inc.
2900 Stender Way
Santa Clara, CA 95054
U.S.A.
www.Mellanox.com

Tel: (408) 970-3400
Fax: (408) 970-3403

Mellanox Technologies Ltd
PO Box 586 Hermon Building
Yokneam 20692
Israel

Tel: +972-4-909-7200
Fax: +972-4-959-3245

Contents

7 MPI20

1 Overview

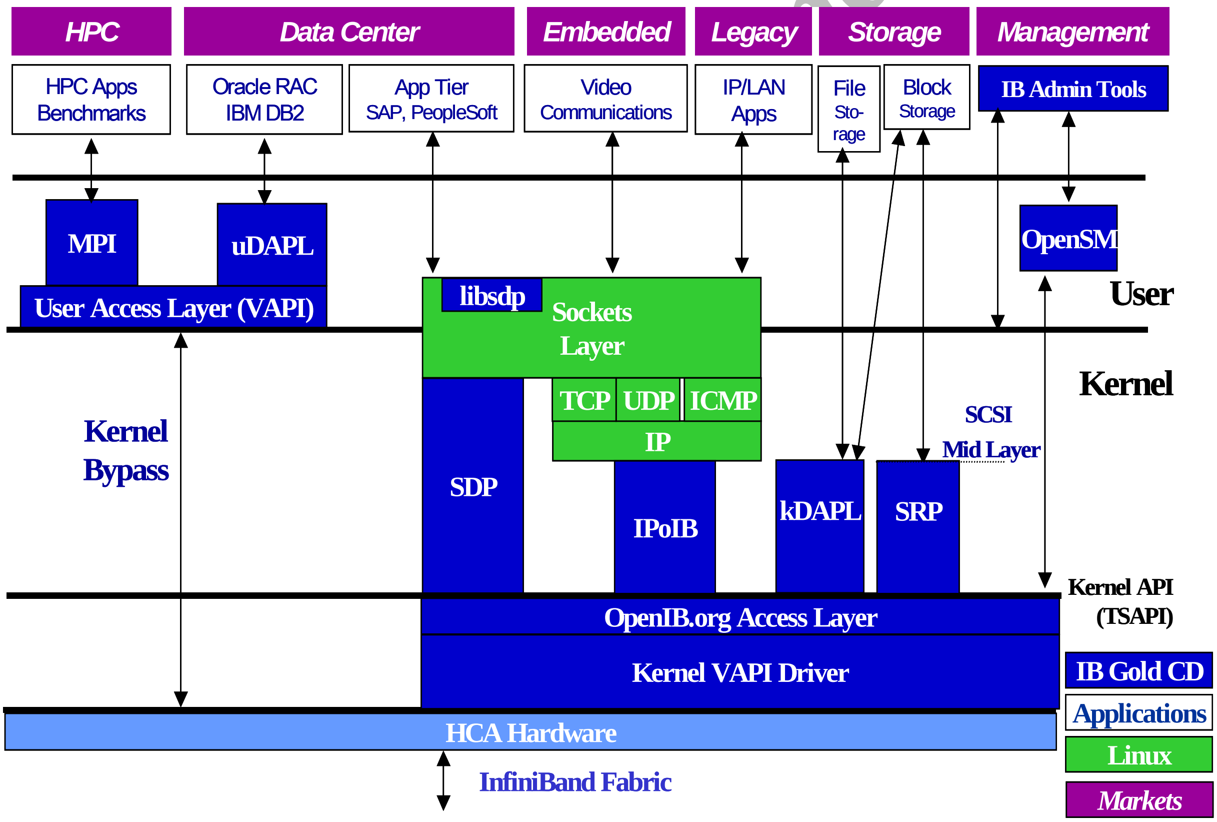
1.1 Introduction

The IB Gold Distribution (IBGD) stack is a full InfiniBand (IB) upper layer protocol (ULP) stack for the Linux operating system. The stack includes: IB drivers, core services, and access layer for the Host Channel Adaptor (HCA) hardware. It also includes the following ULPs:

- IPoIB - IP over IB driver (page 8)
- SDP - Sockets Direct Protocol (page 10)
- SRP Initiator - SCSI RDMA Protocol Initiator (page 13)
- uDAPL - User Direct Access Programming Layer (page 17)
- kDAPL - Kernel-level Direct Access Programming Layer (page 19)
- MPI - Message Passing Interface (page 21)

Figure 1 shows the IBGD stack and how ULPs interface with the hardware, and with kernel and user space.

Figure 1: IBGD Stack



The IBGD stack is composed of a verbs interface (VAPI) and core services. The core services include: management interface (SMI/GSI), Connection Manager (CM), Subnet Administrator (SA) interface, and Device Management (DM) Agent interface. The stack includes components for both user-mode and kernel applications. The core services run in the kernel and expose an interface to user-mode for verbs, CM and management.

The ULPs included in the package enable efficient communication between processes (MPI and SDP), integration with legacy applications (IPoIB), access to storage (SRP), and generic interface to RDMA fabric (DAPL).

This manual provides a basic description of the various software components, and covers how to operate and configure the software modules.

1.2 Loading and Unloading the of Stack

The basic components of the stack are automatically loaded when the machine boots. It is also possible to control which of the components is to be loaded at boot, or to manually load/unload the entire stack.

> Note: OpenSM is not covered in this document. For details on OpenSM, refer to *OpenSM User's Manual, Document no. 2277UM*.

The file /etc/infiniband/openib.conf controls the load-able modules in the stack at boot time. Modify this file if you wish to change the settings. The following is the default configuration for X86 platforms:

```
# Start HCA driver upon boot
ONBOOT=yes

# Load IPoIB
IPOIB_LOAD=yes

# Load sdp module
SDP_LOAD=yes

# Load kDAPL module
KDAPL_LOAD=yes

# Load SRP module
SRP_LOAD=no

# Load SRP target module
SRP_TARGET_LOAD=no
```

To *manually load* the stack, run: `/etc/init.d/openibd start`

To *manually unload* the stack, run: `/etc/init.d/openibd stop`

Note that these load/unload operations control only the modules that are configured in openib.conf.

The files /etc/infiniband/ifcfg-ib0 and /etc/infiniband/ifcfg-ib1 control parameters (such as IP address, NetMask) of the ib0 and ib1 network devices. Although it is best to configure those parameters during the installation process, it is possible to be edit the files at any time.

1.3 Subnet Manager

All ULPs require a proper operation of a Subnet Manager (SM) running on the fabric. An SM can run on any node or on an IB switch. OpenSM is a Subnet Manager provided as part of the IBGD stack. For further details on OpenSM, refer to the OpenSM User's Manual.

> Note: The SM is required to run at all times, not only for initial fabric setup.

1.4 Debugging the Stack

The stack contains a /proc file-system which allows status monitoring. Following is an example of such a very useful file. Other /proc file-system files are described in each ULP section below.

Example: To view the state of a port and its other attributes (such as the LID), enter:

```
cat /proc/infiniband/core/ca1/port1/info
```

If the port is operating properly, its state should be reported as “ACTIVE”.

Error messages are logged into /var/log/messages. It is also possible to activate the stack in debug mode, where logging is much more verbose. In order to control verbosity of debug log messages into /var/log/messages use the command:

```
echo “<mod_id> <verbosity> 0xFFFFFFFF” > /proc/infiniband/tracelevel
```

where:
<mod_id> is the module number (use “cat /proc/infiniband/tracelevel” to capture the relevant module ID)
<verbosity> is a number in the range 0..5 (0- quiet, 5- very verbose)

For debug of the VAPI driver and further information, refer to:

- The README file under the IBGD installation directory (see ‘Starting and Verifying the IB Fabric’)
- VAPI (HCA Driver) Release Notes
- FW-23108 InfiniHost Firmware Release Notes
- FW-25208 InfiniHost III Ex (acting as an InfiniHost) Firmware Release Notes
- FW-25218 MemFree InfiniHost III Ex Firmware Release Notes

2 IP Over IB

2.1 Overview

The IP over IB (IPoIB) driver encapsulates IP datagrams over InfiniBand messages. The driver mimics an Ethernet device and behaves the same from the point of view of the operating system. The encapsulation process adds to the IP datagrams an encapsulation header, and sends the outcome over UD transport service. The interface supports unicast, multicast and broadcast.

The following two documents (drafts) cover IPoIB specification, and they can be downloaded from www.ietf.org:

- draft-ietf-ipoib-architecture-04
- draft-ietf-ipoib-ip-over-infiniband-07

2.2 Configuring IPoIB

The IPoIB configuration is similar to any other network device. Mellanox IB devices support two IPoIB ports named ib0 and ib1.

If the IBGD stack is configured to automatically load IPoIB upon boot and you wish to change the IP addresses of IPoIB ports, edit the files: /etc/infiniband/ifcfg-ib0 and /etc/infiniband/ifcfg-ib1.

2.3 Activating IPoIB

As mentioned before, it is possible to configure the IBGD stack to automatically load the IPoIB driver upon boot. It is also possible to manually load and unload the IPoIB driver.

To bring interfaces up, use:

```
ifconfig  ib0  <IP Address>
ifconfig  ib1  <IP Address>
```

The user can open either of these ports and use both of them.

To bring interfaces down, use:

```
ifconfig ib0 down
ifconfig ib1 down
```

2.4 Debugging IPoIB

The IPoIB driver can be easily checked by a simple ping to any other node in the subnet (using the IPoIB IP address). For example:

```
ping  <remote IPoIB IP address>
```

While the device is running, it is possible to get statistics from the device through the ifconfig command. Statistics include: Rx/Tx packets, Rx/Tx errors, Rx/Tx dropped, Rx/Tx overruns, and Rx/Tx bytes received/sent. For example:

```
ifconfig ib0
```

Other device info can be obtained through the /proc file-system. For each port, there are two tables:

1. IPoIB Pseudo ARP table: /proc/infiniband/ipoib_arp_ibX
2. IPoIB multicast table: /proc/infiniband/ipoib_mcast_ibX

3 Sockets Direct Protocol

3.1 Overview

Sockets Direct Protocol (SDP) is a byte-stream transport protocol that closely mimics TCP's stream semantics. SDP utilizes InfiniBand's advanced protocol offload, and zero copy capabilities. Because of this, SDP can have lower CPU and memory bandwidth utilization when compared to conventional implementations of TCP, while preserving the familiar byte-stream oriented semantics upon which most current network applications depend.

SDP can be used by applications and improve their performance transparently (that is, without any recompilation). Since SDP has the same socket semantics as TCP, an existing application are able to run using SDP. The difference is that its TCP socket gets replaced with an SDP socket.

It is also possible to explicitly set up the sockets of applications to be SDP. To do this it is necessary to select AF_INET_SDP as the socket family.

The SDP protocol is composed of a kernel module that implements the SDP as a new address-family/protocol-family, and a library that is used for replacing the TCP address family with SDP according to a policy.

SDP shares the same IP address with the IPoIB driver and uses the same ARP mechanism to discover neighbors.

The SDP specification is described in “Annex A4” of the *InfiniBand Architecture Specification, Volume 1, Release 1.1*.

3.2 Libsdp.so Library

Libsdp.so is a dynamically linked library, which is used for transparent integration of applications with SDP. The library is preloaded, and therefore takes precedence over glibc for certain socket calls. Thus, it can transparently replace the TCP socket family with SDP socket calls.

The library also implements a user-level socket switch. Using a configuration file, the system administrator can set up the policy that selects the type of socket to be used. Libsdp also has the option to allow server sockets to listen on both SDP and TCP interfaces. The various configurations with SDP/TCP sockets are explained inside the ‘libsdp.conf’ file.

3.3 Environment Variables

For the transparent integration with SDP, two environment variables are required. The environment variables are used for preloading libsdp and configuring the policy:

- **LD_PRELOAD** - This environment variable should point to the libsdp.so library. The variable should be set by the system administrator to ${prefix}/lib/libsdp.so[1].
- **LIBSDP_CONFIG_FILE** - This environment variable must point to the libsdp.conf file. By default, it points to: ${prefix}/etc/libsdp.conf.

In order to avoid setting LD_PRELOAD, it is possible to add lbsdp.so into /etc/ld.so.preload. This causes the library to be preloaded into any executable.

For non-transparent integration with SDP, no special environment variable is necessary. It is only required to recompile the application replacing AF_INET with AF_INET_SDP. The constant AF_INET_SDP is defined in sdp_inet.h.

---

[1] ${prefix} is set at package installation time. By default, it is /usr/local/ibgd.

3.4 Policy

Libsdp.conf is the policy file that controls transparent replacement of TCP sockets with SDP sockets. There are rules to control the server and rules to control the client side. If a rule is matched, then an SDP socket replaces the TCP socket.

Client rules:

```
match destination <ip_port>
where:
 <ip_port> is <ip_addr>[/<prefix_length>][:<start_port>[-<end_port>]]
```

Server rules:

```
match listen <ip_port>
where:
 <ip_port> is <ip_addr>[/<prefix_length>][:<start_port>[-<end_port>]]
```

General rules:

```
match <program_name>
where:
 <program_name> is a string representing a program name (wildcards allowed)
```

For example:

```
match listen *:5001 program ttcp
```

3.5 Activating SDP

SDP module can be automatically loaded when the machine boots, see Section 1.2, "Loading and Unloading the of Stack," on page 6. In order to manually load the SDP module use:

```
modprobe ib_sdp
```

And to manually unload:

```
modprobe -r ib_sdp
```

When the two environment variable stated above are defined, and the configuration within libsdp.conf is set up, all you have to do is run any socket application that uses AF_INET sockets. According to the policy, SDP will be selected and the data will flow through SDP.

In short:

```
system administrator configures libsdp.conf
export LD_PRELOAD=${prefix}/lib/libsdp.so
export LIBSDP_CONFIG_FILE=${prefix}/etc/libsdp.conf
run application
```

Another option is to recompile the application using AF_INET_SDP socket, in which case application will run SDP socket regardless of any configuration.

Note that SDP uses the same IP address space that the IPoIB driver uses. It therefore required that the client and server of the SDP application will be on the same subnet where IPoIB run. If your computer has an another NIC, you need to make sure that the SDP runs on the IB fabric, i.e. the server address must be on the IPoIB subnet.

3.6 Debugging SDP

In order to list the active and listening connections on SDP use:

```
cat /proc/infiniband/sdp/opt_conn
```

In order to log libsdp debug message to a file, edit libsdp.conf and add (for example) the following:

```
log min-level 5 destination file <filename>
```

Further information about log options can be found in libsdp.conf.

4 SRP Initiator

4.1 Overview

SRP (SCSI RDMA Protocol) is designed to take full advantage of the protocol offload and RDMA features provided by the InfiniBand Architecture. SRP allows a large body of SCSI software to be readily used on InfiniBand Architecture, and is rapidly emerging as the protocol of choice for block-based storage.

The SRP device driver differs from traditional low-level SCSI drivers in Linux. The SRP driver does not control a local HBA. Instead, it controls a connection to an IO controller to provide access to remote storage devices across an InfiniBand fabric. The IO controller resides in an IO unit and provides storage services.

The SRP device driver is known as the SRP Initiator whereas the service on the InfiniBand IO controller is known as the SRP Target.

The SRP protocol provides transport services to enable a basic client-server model where an initiator presents SCSI tasks to a target for execution. A typical SRP IO transaction is as follows:

- The initiator builds an SRP request message that contains a SCSI command, a device logical unit number, and a data buffer memory descriptor (or scatter/gather list). It then sends the request to the target.
- For read requests, the target receives the SRP request, reads the data from disk/cache and performs an RDMA operation to transfer the data into the initiator data buffer.
- For write requests, the target receives the SRP request and performs an RDMA operation to transfer the initiator data buffer contents to the target, then it commits it to the disk.
- The target builds an SRP response message indicating the completion status of the request, and sends the response to the initiator.

SRP 1.0 specification is available at www.t10.org.

4.2 Environment Setup

For basic operations no variable needs to be set.

By default, Linux does not support SCSI devices with multiple LUNs. If SRP Target has multiple LUNs (IOC contains multiple LUNs), then the kernel needs to be configured to support that. There are two options to enable multiple LUN support:

- Add “max_scsi_luns=n” to /etc/modules.conf and rebuild the initrd. (In Kernel 2.4 and RH distribution).
- Rebuild the kernel with CONFIG_SCSI_MULTI_LUN=y in the .config file.

4.3 Activating SRP

SRP requires an SRP Target to be attached to the InfiniBand fabric. The default setup does not automatically load the SRP Initiator upon boot.

To *manually load* the SRP Initiator use:

```
modprobe ib_srp
```

To *manually unload* the SRP Initiator use:

```
modprobe -r ib_srp
```

Once the SRP Initiator is up,SRP Target new SCSI block devices are available through:

- /dev/sdX for a device without partitions, where X stand for a letter: a,b,c, etc
- /dev/sdXY for a device with partitions, where Y stand for a number: 1,2 etc (and defines partition number)

The drive letter (X) is allocated dynamically by Linux as the SCSI device is discovered.

It is preferred that the SRP Target will be up and running prior to loading the SRP Initiator. If the Initiator is loaded before the target than after the SRP Target is loaded, run : rescan-scsi-bus.sh script to detect the Target.

4.4 Debugging SRP

There are certain tools and utilities that can be used to debug and understand the SRP Initiator attachment to the Target. These are:

- SCSI /proc file-system:

  cat /proc/scsi/scsi -lists all active SCSI devices. SRP Targets are added to the list as they are discovered

  The following is an example of a listed SRP Target (at channel 0, ID 1, LUN 0):

  ```
  Attached devices:
  Host: scsi0 Channel: 00 Id: 01 Lun: 00
    Vendor: Mellanox Model: IBSRP10-TGT      Rev: 1.46
    Type:   Direct-Access ANSI SCSI          revision: 03
  ```

- SRP /proc file-system:

  cat /proc/scsi/srp/x- indicates the active connections to IO Controllers with their status. ‘x’ represents the device (host) number (it is incremented each time the device driver is reloaded)

- To view all SCSI devices and to create/delete partitions on the disk use fdisk:

  ```
  fdisk -l          - lists all devices/partitions
  fdisk /dev/sdX   - creates/deletes partitions
  ```

- To create a file-system and mount it use (this is an example):

  ```
  mke2fs /dev/sdXY
  mount   /dev/sdXY <directory>
  ```

- In order to load the SRP Initiator in debug mode, where various debug messages get echoed into /var/log/messages, use:

  ```
  modprobe ib_srp srp_tracelevel=<level>
  ```

  where <level> shall be 0 (quiet) to 5 (verbose)

4.5 Additional Features

This section presents the following features: “Persistent Target Binding” and “LUN Masking”.

4.5.1 Persistent Target Binding

Since Linux assigns SCSI device nodes dynamically as each additional SCSI logical unit is detected, the mapping from device nodes (e.g., /dev/sda or /dev/sdb) to SRP SCSI targets and logical units may vary.

Variations in process scheduling and network delay may result in SRP SCSI targets being mapped to different SCSI device nodes each time the driver is started. Hence, if applications or operating system utilities are configured to use the standard SCSI device nodes to access SRP SCSI devices, it is possible for SCSI commands to address the wrong targets or logical units.

To provide a more reliable name-space, the SRP SCSI driver uses its own package to create persistent device naming for SRP SCSI devices. The SRP SCSI devices can be found under the /dev/srp/ directory in one of the following formats:

```
srp-<IOC_GUID>-l<lun_id>
srp-<IOC_GUID>-l<lun_id>p<partition>
```

Examples:

```
/dev/srp/srp-0000c900012a3800-l0
/dev/srp/srp-0000c900012a3800-l0p1
/dev/srp/srp-0000c900012a3800-l0p2
```

These device names (symbolic links) point to the block devices under the /dev/scsi/ directory in the following format:

```
sdh<host_num>-0c0i<target_id>l<lun_id>p<partition>
```

By using this device naming method, it is ensured that:

1. The same SRP SCSI bus and target id number are used for a particular SRP SCSI Target name in every SRP SCSI session.
2. A Linux SCSI target is always mapped to the same physical storage device after each machine boot.
3. The SCSI device names used at device creation are always mapped to the same SRP SCSI target.

This method of device naming results in persistent device mapping and is termed the Persistent Target Binding feature. To avoid errors in mapping device names, the device names it creates should be used by applications and fstab files, and not those of direct referencing of particular SCSI device nodes.

Activation. The Persistent Target Binding feature is activated automatically upon loading the ib_srp module. Loading the SRP Initiator (with the persistent binding daemon) is done using ‘modprobe ib_srp’. To unload it, see Table 1.

Table 1 - Unloading the SRP Initiator with Persistent Binding Daemon

For Kernel...	Unload the SRP Initiator using...
2.6	modprobe -r ib_srp
2.4	One of two methods: 1. Run ‘remove_srp_bind_persistent.sh’ then run ‘modprobe -r ib_srp’ 2. Run ‘openib stop’

4.5.2 LUN Masking

This feature allows binding and/or masking of specific partitions in one machine, and binding and/or masking of other partitions in another machine.

The srp_persistent_bind.conf configuration file acts as the vehicle for manipulating partition binds and masks. It is located under ${prefix}/etc/[1].

By default, the SRP Initiator binds all the partitions it finds *unmasked* during the discovery stage.

Partition Binding:

To bind a partition, add a line in the srp_persistent_bind.conf file with the letter ‘b’ as the line header, followed by an SRP partition name of the format: srp-<IOC_GUID>-l<lun>p<partition>.

Example:

```
b srp-0002c901093dc560-l0p2
```

Notes:

1. No spaces are allowed before the letter ‘b’.
2. Only *one space* is allowed between the letter ‘b’ and the SRP partition name.

Partition Masking:

To mask a partition, add a line in the srp_persistent_bind.conf file with the letter ‘m’ as the line header, followed by an SRP partition name of the format: srp-<IOC_GUID>-l<lun>p<partition>.

Example:

```
m srp-0002c901093dc560-l0p2
```

To mask all partitions that are *not* bound, enter:

```
m srp-ALL
```

Notes:

1. No spaces are allowed before the letter ‘m’.
2. Only *one space* is allowed between the letter ‘m’ and the SRP partition name.

Example of an srp_persistent_bind.conf file:

```
###################################
# Partitions to bind
b srp-0002c901093dc560-l0p2
# Partitions to mask
m srp-ALL
###################################
```

---

[1] ${prefix} is set at package installation time. By default, it is /usr/local/ibgd.

5 uDAPL

5.1 Overview

The uDAPL (User Direct Access Programming Library) is a generic programming interface to RDMA capable transports. The library exposes a well defined API for applications that require access to RDMA-capable devices. The specification and API are defined by the DAT Collaborative (www.datcollaborative.org) and the stack is based on the Source Forge DAPL code.

The API is designed for SW applications, therefore is not as low level as other APIs such as the access layer or VAPI may be. It is going to remain preserved regardless of the underlying layers (HW and SW components).

The uDAPL component of the IBGD stack provides user space libraries that may be dynamically or statically loaded for the DAT collaborative API.

5.2 Activating uDAPL

Before running a DAT-based application, the relevant IBGD stack module should be loaded. This can be done automatically by entering: /etc/init.d/openibd start

5.3 Compiling, Linking, and Running uDAPL

- In SW code that uses uDAPL, the following DAT header should be included:

  #include <dat/dat.h>
- When compiling, the relevant include path should be listed:

  gcc ... -I${prefix}/include[1] ...
- When linking the DAT and DAPL libraries, the following should be included:

  -L${prefix}/lib -ldat -ldapl

- When running the application, the libraries are loaded from ${prefix}/lib. Therefore ${prefix}/lib must be included in the library path.

5.4 Debugging uDAPL

To verify that the required modules are loaded, perform the following steps:

1. Check that the ib_udapl is loaded by entering:
   ```
   /sbin/lsmod | grep ib_dapl_srv
   ```
2. Check that the InifniBand IPoIB ports are up using:
   ```
   /sbin/ifconfig ib0
   /sbin/ifconfig ib1
   ```

- Set the environment variable DAPL_DEBUG to ‘1’ then compile uDAPL.
- To enable debug features, set the two environment variables DAT_DBG_TYPE and DAPL_DBG_TYPE according to TABLE TBD below.

---

[1] ${prefix} is set at package installation time. By default, it is /usr/local/ibgd.

Table 2 - DAPL Debug Level Setting for dat_registry

Debug Type	**DAT_DBG_TYPE =**
DAT_OS_DBG_TYPE_ERROR	0x1
DAT_OS_DBG_TYPE_GENERIC	0x2
DAT_OS_DBG_TYPE_SR	0x4
DAT_OS_DBG_TYPE_DR	0x8
DAT_OS_DBG_TYPE_PROVIDER_API	0x10
DAT_OS_DBG_TYPE_CONSUMER_API	0x20
DAT_OS_DBG_TYPE_ALL	0xFF

Table 3 - DAPL Debug Level Setting for uDAPL

Debug Type	**DAPL_DBG_TYPE =**
DAPL_DBG_TYPE_ERR	0x0001
DAPL_DBG_TYPE_WARN	0x0002
DAPL_DBG_TYPE_EVD	0x0004
DAPL_DBG_TYPE_CM	0x0008
DAPL_DBG_TYPE_EP	0x0010
DAPL_DBG_TYPE_UTIL	0x0020
DAPL_DBG_TYPE_CALLBACK	0x0040
DAPL_DBG_TYPE_DTO_COMP_ERR	0x0080
DAPL_DBG_TYPE_API	0x0100
DAPL_DBG_TYPE_RTN	0x0200
DAPL_DBG_TYPE_EXCEPTION	0x0400
ALL	0xFFFF

5.5 dapltest

The IBGD package includes the dapltest which is a uDAPL test in the SourceForge repository.

To run the test enter the command: ${prefix}[1]/bin/dapltest

For a full description of the test and optional flags see under ${prefix}/docs, or under the DAPL project directories in the SourceForge repository.

---

[1] ${prefix} is set at package installation time. By default, it is /usr/local/ibgd

6 kDAPL

6.1 Overview

The kDAPL (Kernel Direct Access Programming) is a generic programming interface to RDMA capable transports. DAPL exposes a well defined API for applications that require access to RDMA-capable devices. The specification and API are defined by the DAT Collaborative (www.datcollaborative.org) version 1.1 and the stack is based on the Source Forge DAPL code.

The API is designed for SW applications, therefore is not as low level as other APIs such as the access layer or VAPI may be. It is going to remain preserved regardless of the underlying layers (HW and SW components).

The kDAPL component of the IBGD stack exports the kDAPL symbols in the kernel.

6.2 Activating kDAPL

Before running a DAT-based application, the relevant IBGD stack module should be loaded. This can be done as follows:

- Edit the file /etc/infiniband/openib.conf (see Section 1.2, "Loading and Unloading the of Stack," on page 6) to have the following parameter setting: KDAPL_LOAD=yes
- Reload the IBGD stack by typing /etc/init.d/openibd restart

6.3 Compiling, Linking, and Running kDAPL

- In SW code that uses kDAPL, the following DAT header should be included:

```
#include <dat/kdat.h>
```

- When compiling, the relevant include path should be listed:

```
gcc ...  -I${KERNEL_SRC_TREE}/drivers/infinband/ulp/kdapl/dat/include...
```

- Before the modules that use kDAPL can be called by 'insmod', the following modules should be loaded: 'dat_registry' and 'ib_kdapl'.

6.4 Debugging kDAPL

- Before installing IBGD, set the environment variable KDAPL_DEBUG to '1' in order to compile kDAPL with debug features.
- In order to display debug prints of kDAPL (assuming that the stack was compiled with kDAPL debug prints enabled), unload the 'ib_kdapl' and 'dat_registry' modules.
- The debug prints will be in /var/log/messages.
- Run 'insmod' on the 'dat_registry' module with the parameter DbgLvl set according to Table 4 below.

Table 4 - kDAPL Debug Level Setting for dat_registry Module

Debug Type	DbgLvl =
DAT_OS_DBG_TYPE_ERROR	0x1
DAT_OS_DBG_TYPE_GENERIC	0x2
DAT_OS_DBG_TYPE_SR	0x4

Table 4 - kDAPL Debug Level Setting for dat_registry Module (Continued)

Debug Type	DbgLvl =
DAT_OS_DBG_TYPE_DR	0x8
DAT_OS_DBG_TYPE_PROVIDER_API	0x10
DAT_OS_DBG_TYPE_CONSUMER_API	0x20
DAT_OS_DBG_TYPE_ALL	0xFF

- Run ‘insmod’ on the ‘ib_kdapl’ module with the parameter DbgLvl set according to Table 4 below.

Table 5 - kDAPL Debug Level Setting for ib_kdapl Module

Debug Type	DbgLvl =
DAPL_DBG_TYPE_ERR	0x0001
DAPL_DBG_TYPE_WARN	0x0002
DAPL_DBG_TYPE_EVD	0x0004
DAPL_DBG_TYPE_CM	0x0008
DAPL_DBG_TYPE_EP	0x0010
DAPL_DBG_TYPE_UTIL	0x0020
DAPL_DBG_TYPE_CALLBACK	0x0040
DAPL_DBG_TYPE_DTO_COMP_ERR	0x0080
DAPL_DBG_TYPE_API	0x0100
DAPL_DBG_TYPE_RTN	0x0200
DAPL_DBG_TYPE_EXCEPTION	0x0400
ALL	0xFFFF

6.5 kdapltest

The IBGD package includes the kdapltest which is a kDAPL test in the SourceForge repository.

To run the test:

1. run ‘modprobe kdapltest’ in order to load the test module to the kernel.
2. Run the test using the command: ${prefix}[1]/bin/kdapltest

For a full description of the test and optional flags see under ${prefix}/docs , or under the DAPL project directories in the SourceForge repository.

Note: After the test ends, the user needs to unload the kdapltest module using ‘modprobe -r kdapltest’.

---

[1] ${prefix} is set at package installation time. By default, it is /usr/local/ibgd.

7 MPI

7.1 Overview

The Mellanox IB Gold Distribution includes two MPI packages:

1. MPI channel implementation over InfiniBand from Ohio State University (OSU) (http://nowlab.cis.ohio-state.edu/projects/mpi-iba/)
2. MPI channel implementation over InfiniBand from the National Center for Supercomputing Applications (NCSA) (http://vmi.ncsa.uiuc.edu/)

It is possible to choose to install one or both of these packages. Both packages are described in the following sections.

7.2 OSU MPI

7.2.1 Installation Directories and Environment Setup

OSU MPI implemented over InfiniBand is installed by default under ${prefix}/mpi/osu/mvapich-0.9.4[1].

The MPI applications (presta, pallas and osu_tests) can be found under ${prefix}/mpi/osu/tests/.

It is recommended to set the following environment variables to facilitate the usage of OSU MPI software:

```
export MICH_ROOT=${prefix}/mpi/osu/mvapich-0.9.4
export PATH=${MPICH_ROOT}/bin:${PATH}
```

Please note that OSU MPI is pre-configured to use "ssh" to launch MPI jobs on the cluster machines, so password-less "ssh" should be configured from the main node to all other cluster nodes.

It is also possible to launch MPI jobs using "rsh". To do so, use mpirun_rsh with the "-rsh" option (see below). As above, password-less "rsh" should be configured from the main node to all other cluster nodes.

7.2.2 Compiling and Running MPI Programs

1. Compile using mpicc or mpif77 compilers located under $MPICH_ROOT/bin.

   For example: `$MPICH_ROOT/bin/mpicc -o cpi cpi.c`
2. Use "mpirun_rsh" to run mpi programs: `$MPICH_ROOT/bin/mpirun_rsh -np N h1 h2 ... hN a.out args`

   For example: `mpirun_rsh -np 4 node1 node2 node3 node4 ./cpi`

   Or `mpirun_rsh -hostfile machines -np 4 ./cpi`

   Where "machines" is a text file defining the list of machines in the cluster. For the example above this file will look as follows:

   ```
   node1
   node2
   node3
   node4
   ```

---

[1] ${prefix} is set at package installation time. By default, it is /usr/local/ibgd.

Note that it is possible to launch MPI using rsh by adding the option "-rsh" to the mpirun_rsh command line. So the examples above become:

7.2.3 Additional Documentation

For additional information, see the following files:

- ${prefix}/docs/README_MVAPICH
- ${prefix}/docs/mvapich.tuning_guide
- ${prefix}/docs/mvapich.user_guide

Please email your feedback to: mvapich_help@cis.ohio-state.edu

7.3 NCSA MPI

7.3.1 Installation Instructions

Please read these VMI2/MPICH-VMI instructions if you choose to build and install MPI from the source code packages and RPMs provided by the Mellanox IB Gold Distribution. If you choose to build and install VMI2/MPICH-VMI with different configuration options please refer to the NCSA web site.

Prerequisites:

VMI2 and MPICH-VMI depend on several packages to build and function properly. If these are not present on your system, please install them before proceeding.

The required packages:

- autoconf 2.52 or later
- automake 1.5 or later
- libtool 1.4.2 or later
- cURL 7.8 or later
- expect 1.95 or later

The following is an optional package:

- Mesa (any recent version)

Binary install via RPMs:

If you use the binary RPMs provided on the IB Gold Distribution, it is possible to skip compiling the VMI2, MPICH-VMI source code. However, please be advised that the RPMs cannot be relocated, and will always be installed under ${prefix}/mpi/ncsa.

Currently VMI2 and MPICH-VMI consist of the following RPMs:

- vmi_mlx-2.0.1.i386.rpm - This RPM contains the VMI runtime/development libraries, applications and headers. This RPM should be installed on all the nodes in the cluster.
- mpich_mlx-ch_vmi-2.0.1.i386.rpm - This RPM contains the MPICH-VMI runtime/development libraries, support scripts and headers. This RPM should be installed on all the nodes in the cluster.

There are several things to keep in mind when installing the RPMs. It has been reported that several newer versions of the cURL RPMs do not link "/usr/lib/libcurl.so" to "/usr/lib/libcurl.so.1". This may cause a failure when attempting to run VMI binaries that come from RPMs. If this happens try adding the link manually.

7.3.2 VMI2/MPICH-VMI Configuration Instructions

Running VMI Daemons:

You will need to start a copy of the 'vmieyes' daemon on all the nodes that can run a MPI process. You can either do this via the init script bundled with the VMI runtime package (/etc/init.d/vmieyes start), or manually. The vmieyes daemon is installed under $VMI_INSTALL_PATH/sbin directory.

It is recommended, though not necessary, to run the vmieyes daemon as root. The vmieyes daemon is used only during job startup and hence does not consume any significant computing resources when running. It is recommended that the vmieyes daemon be chkconfig to start automatically on reboot. MPICH-VMI jobs will fail to run if a node does not have the vmieyes daemon running on it.

MPD Ring Setup:

There are two files which need to be propagated to every node in a given mpd ring:

1. /etc/mpd.conf

   Required configuration: copy to all nodes upon installation. /etc/mpd.conf is populated with a random password string. This needs to be common among all the nodes in an mpd ring. You can take any node's version of this file and copy it to all the others. It does not matter which one, as long as they are the same.

2. /etc/init.d/mpd

   Required configuration:
     set MPD_LISTENER_HOST (default not set, must be the same on all nodes)

   Optional configuration:
   set EXEC_ON_LISTENER (default = 1)

   Optional configuration:
   set MPD_PORT (default = 666)

   Required configuration:
   copy to all nodes

This is the rc init script for the mpd daemon. In an mpd ring, all mpds are launched to connect to a "listener host". Whichever node you specify as the listener host will automatically be launched in the listener mode. The default setting *is* to allow execution on the listener host. e.g., if your head node is operating as the listener.Any configuration changes need to be made before the init script is propagated of course.

Note: The order of mpd daemon start is important (unlike the vmieyes daemon).

The "listener host" daemon must me started first, before all mpd daemons have been started. Once all mpd daemons are started, you can check if all the machines are connected by: /etc/init.d/mpd hosts

Please check that all the requested machines are listed.

Default machines file: mpich_vmi2_dir/share/machines.list

Optional configuration: list hostnames

The machines.list file may be populated with a list of hosts to use when the -machinefile option is not used on the mpirun command line. Format of each line is one of the following:

```
HOSTNAME

-or-

HOSTNAME:[num_procs]
```

Defaults in mpirun: mpich_vmi2_dir/bin/mpirun.ch_vmi

Optional configuration: set values for custom settings at the top of the file, the following set of variables is defined. Those prefixed by "DF_" are only set if the value is not set explicitly. e.g., if -specfile is not specified on the mpirun command line, then the value in $DF_VMI_SPECFILE is used:

```
LD_LIBRARY_PATH=$VMI_INSTALL_PATH/lib:$LD_LIBRARY_PATH

DF_rshcmd=ssh

DF_LOG_ENABLE=0

DF_LOGFILE=$MPIRUN_HOME/../log/mpirun.ch_vmi.log

DF_LOG_DIRECTIVES=$MPIRUN_HOME/mpirun.ch_vmi.logger

DF_machineFile=$MPIRUN_HOME/../share/machines.list

DF_VMI_SPECFILE=$VMI_INSTALL_PATH/specfiles/mst.xml

DF_VMI_SPECFILE_PATH=$VMI_INSTALL_PATH/specfiles

DF_VMI_LAUNCHER=1          # Detected launcher overrides this value

DF_VMI_VERBOSE=0
```

Logging mpirun:

By default, mpirun logging is disabled. This can be changed globally in mpirun.ch_vmi (above), but can still be force-disabled on the mpirun command line with the -nolog option. The default logfile is also set at the top of mpirun.ch_vmi (above). Logging is mainly useful to sysadmins for debugging and tracking mpirun use. It is quiteverbose, and if enabled, the size of the logfile should be monitored.

7.3.3 VMI2/MPICH-VMI Usage Instructions

Running code with MPICH-VMI:

To compile a code with MPICH-VMI, the standard "mpicc", "mpiCC" and "mpif77" scripts should be used.

"mpirun" is used to launch jobs. Their default location is under: ${prefix}/mpi/ncsa/mpich-vmi-2.0.1/bin

The mpirun that ships with MPICH-VMI supports launching jobs via VMPD/ssh/rsh. In addition to the standard "mpirun" options, MPICH-VMI has the following additional options to better accommodate VMI2.

-nolog	Disables admin logging.
-specfile	Specifies xml specfile (Used to switch the underlying transport) Examples: -specfile openib -specfile tcp
-key [string]	Character string address for VMI process. Automatically allocated by mpirun.ch_vmi, except within grid jobs
-use-ssh	Use ssh as default launch mechanism
-force-shell	Overrides detected launch mechanism with shell (ssh)
-grid-procs [#]	Overrides detected VMI_PROCS for separated cluster run time environments (grid environments)
-debugger [gdb] [totalview]	Select debugger to run with
-mmapthreshold	Allocation at which mmap is used. Default is 4 MB.
-rdmachunk	Base chunk for large RDMA transfers used for Rendezvous protocol. Default is 256k
-rdmapipeline	Max number of RDMA chunks in flight. Default is 3.
-eagerlen	Message size at which to switch from short to rendezvous protocol. Default is 16k.
-eagerrunexcount	Maximum number of unexpected short messages before allocating memory for subsequent receives. Default is 16
-v	Verbose level 1 - MPIRUN verbose & VMI startup
-vv	Verbose level 2 - Warning messages
-vvv	Verbose level 3 - Error messages
-vvvv	Verbose level 10 - Excess debug (Everything)

If mpirun is configured fully for the local setup, NONE of these options are mandatory.

> Note: The "-specfile" option takes full paths (and URLS) or simple file prefixes (gm, tcp, etc.). When specifying a prefix, mpirun assumes that you have a corresponding XML file in VMIINSTALLPATH/specfiles

which contains information about the transport(s) which you wish to use. This automatically supports new XML files added to that directory, for example if the site administrator adds a XML specfile called "newtransport.xml", mpirun will support "-specfile newtransport".

Profiling data is used for self-tuning subsequent runs of an application. The data collected includes

- Job duration
- Job size (MPI world size)
- Executable name
- Transport used
- One-way hash of userid (i.e. NOT the userid)
- Communications stats (send/receive counters per rank)