

NDMP

Network Data Management Protocol

Protocol Specification

Document Version: ***1.7.2***

Last Update: ***10/24/96***

1. OVERVIEW.....	4
1.1 MOTIVATION	4
1.2 AUDIENCE	4
1.3 TERMINOLOGY	4
2. ARCHITECTURE.....	5
2.1 ARCHITECTURAL MODEL.....	5
2.2 COMPARISON ARCHITECTURES.....	7
2.3 STATE DESCRIPTION	8
2.4 INTERFACES	9
2.5 MESSAGES FROM BACKUP SOFTWARE TO THE NDMP SERVER	9
2.6 MESSAGES FROM NDMP SERVER TO BACKUP SOFTWARE.....	10
2.7 MESSAGING PROTOCOL	10
2.8 MESSAGE DEFINITIONS.....	10
2.9 HEADER.....	11
2.10 ERROR.....	12
3. INTERFACES.....	13
3.1 CONNECT INTERFACE.....	13
3.1.1 <i>Open Connection</i>	14
3.1.2 <i>Authorization</i>	14
3.1.3 <i>Close Connection</i>	15
3.2 CONFIG INTERFACE.....	15
3.2.1 <i>Get Host Info</i>	15
3.2.2 <i>Get backup Type Attribute</i>	15
3.3 SCSI INTERFACE.....	16
3.3.1 <i>Open SCSI Device</i>	16
3.3.2 <i>Close Device</i>	16
3.3.3 <i>Get SCSI State</i>	17
3.3.4 <i>Set SCSI Target</i>	17
3.3.5 <i>Reset Device</i>	18
3.3.6 <i>Reset Bus</i>	18
3.3.7 <i>Execute CDB</i>	18
3.4 TAPE INTERFACE	19
3.4.1 <i>Open Tape Device</i>	19
3.4.2 <i>Close Device</i>	20
3.4.3 <i>Get Tape State</i>	20
3.4.4 <i>MTIO</i>	21
3.4.5 <i>Write</i>	22
3.4.6 <i>Read</i>	22
3.4.7 <i>Set Record Size</i>	23
3.4.8 <i>Execute CDB</i>	23
3.5 DATA INTERFACE	23
3.5.1 <i>Get Data State</i>	23
3.5.2 <i>Backup</i>	24
3.5.3 <i>Recover</i>	26
3.5.4 <i>Abort</i>	27
3.5.5 <i>Stop</i>	27
3.5.6 <i>Continue</i>	27
3.5.7 <i>Get ENV</i>	28

4. BACKUP SOFTWARE INTERFACES	29
4.1 NOTIFY INTERFACE.....	29
4.1.1 <i>Notify Paused</i>	29
4.1.2 <i>Notify Halted</i>	29
4.1.3 <i>Notify Connect</i>	29
4.2 LOGGING INTERFACE	30
4.2.1 <i>Log</i>	30
4.2.2 <i>Debug</i>	30
4.2.3 <i>File Recovered</i>	30
4.3 FILE HISTORY INTERFACE.....	32
4.3.1 <i>Add Unix Path</i>	32
4.3.2 <i>Add Unix Dir</i>	33
4.3.3 <i>Add Unix Node</i>	33
5. APPENDIX A - NDMP.X PROTOCOL FILE.....	34
 FIGURE 1. SIMPLE CONFIGURATION	5
FIGURE 2. TWO DRIVE CONFIGURATION.....	6
FIGURE 3. JUKEBOX CONFIGURATION	6
FIGURE 4 - BACKUP STATE DIAGRAM	8

1. Overview

1.1 Motivation

The purpose of this protocol is to allow a backup of a host without requiring a full port of the backup software product. The backup and restore solution is partitioned in a way that minimizes the amount of software required on the host with the tape drive attached.

The system vendors need only be concerned with maintaining compatibility with one, well-defined protocol. The backup vendors can place their primary focus on the sophisticated central backup administration software.

This protocol is specifically intended to support tape drives. This protocol is targeted towards backup software and there are extensive references to the tasks of backup and restore. However, the protocol may be used for other applications in the future.

1.2 Audience

This document is intended for use by software developers to implement Network Data Management Protocol. The reader is assumed to be familiar with network protocol specifications and with the general operation of backup software. The user is not expected to have knowledge of internal backup software behavior.

1.3 Terminology

Backup software host

The host on which the backup software daemons and the backup software databases exists. The backup software host may or may not have a tape drive attached

NDMP host

The host which has a tape drive physically attached and can perform local backups to that tape drive using the NDMP.

NDMP server

The virtual state machine on the NDMP host that is controlled using the NDMP protocol. There is one of these for each connection to the NDMP host. This term is used independent of implementation.

2. Architecture

2.1 Architectural Model

The architecture is a client server model and backup software is considered a client to the NDMP server. For every connection between the client on the backup software host and the NDMP host there is a virtual state machine on the NDMP host that is controlled using the NDMP. This virtual state machine is referred to as the NDMP server. Each state machine controls at most one device used to perform backups. The protocol is a set of XDR encoded messages that are exchanged over a bi-directional TCP/IP connection and are used to control and monitor the state of the NDMP server and to collect detail information about the data that is backed up.

In the most simple configuration, backup software will backup the data from the NDMP host to a tape drive connected to the NDMP host.

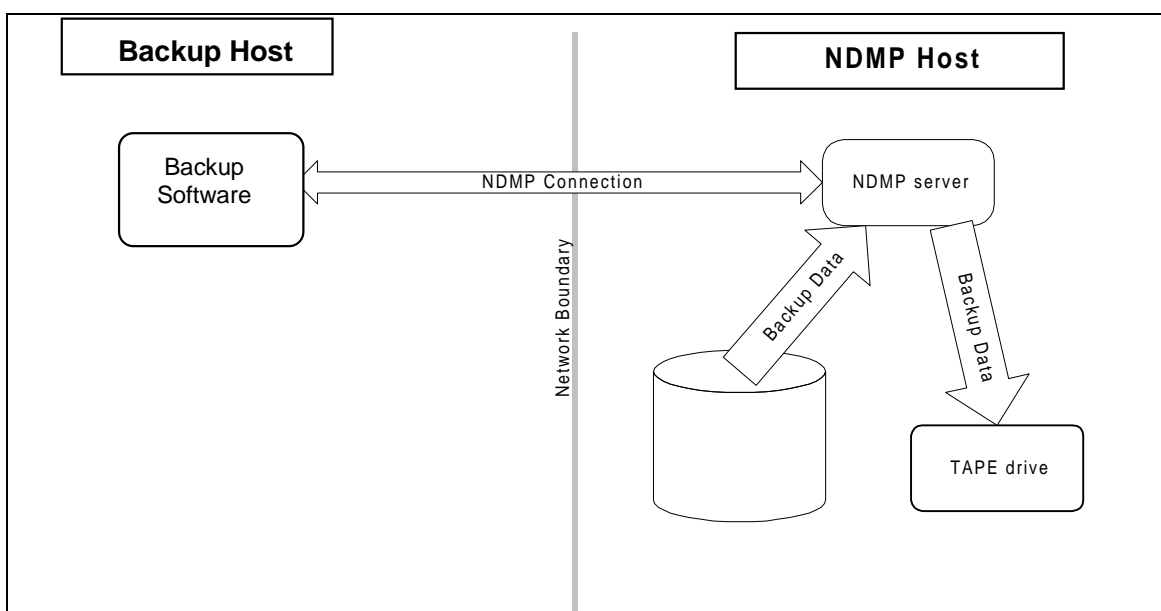


Figure 1. Simple configuration

It is also possible to use the NDMP to simultaneously backup to two tape drives physically attached to the NDMP host. In this configuration there are two instances of the NDMP server on the NDMP host.

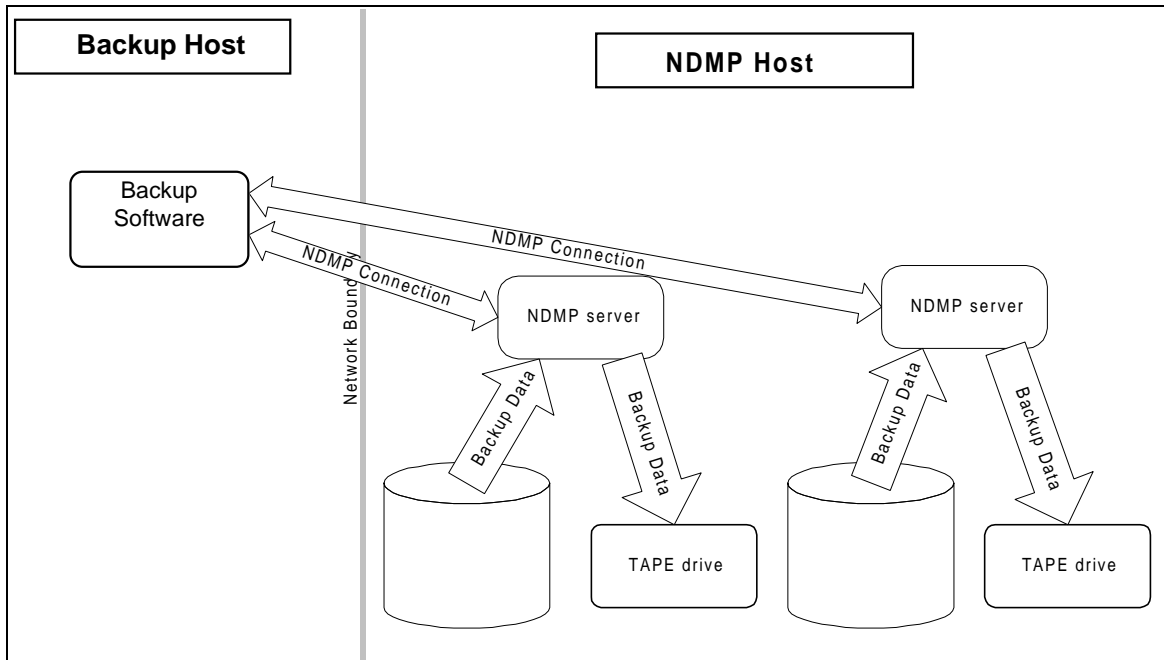


Figure 2. Two drive configuration

The NDMP can be used to backup data to a tape drive in a jukebox that is physically attached to the NDMP host. In this configuration, there is a separate instance of the NDMP server to control the robotic arm in the jukebox.

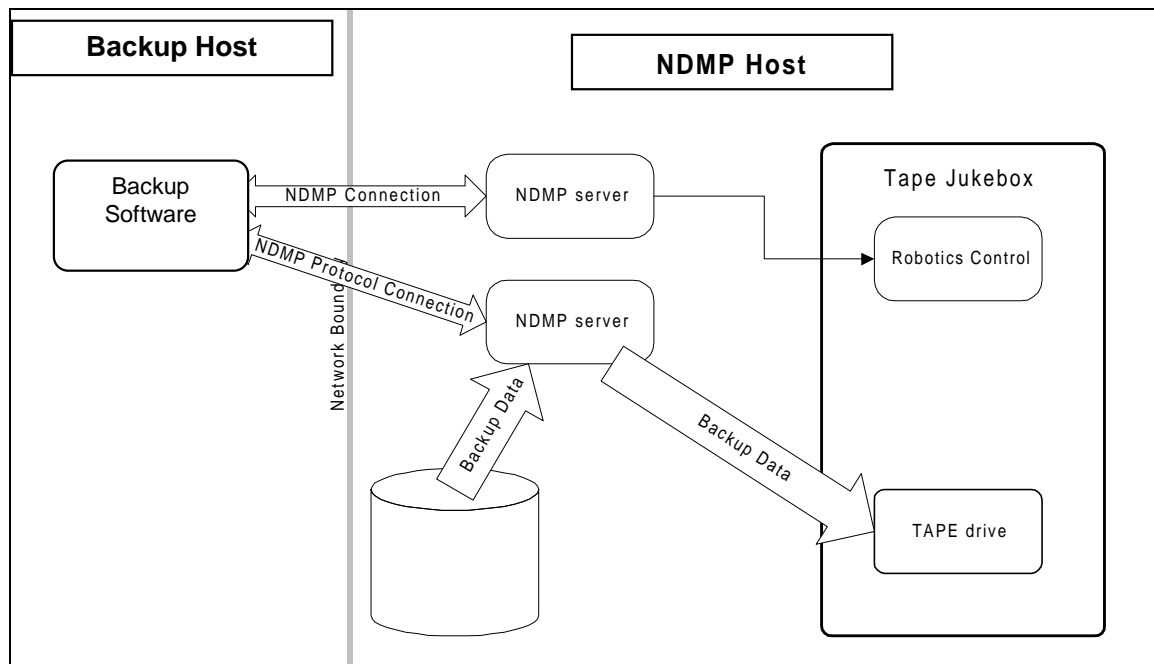


Figure 3. Jukebox configuration

2.2 Comparison Architectures

It is useful to compare the NDMP architecture to other architectures and note the similarities and differences.

rmt

The architecture is similar to the rmt architecture in that connection is made to a generic server and the server is instructed to open a specific tape drive device. The NDMP differs in that it uses a TCP/IP connection to a dedicated port whereas rmt uses the rsh demon to launch a server.

X11

The architecture is similar to the X11 architecture in that it uses a single connection to a TCP/IP port, however it differs in that the NDMP server is not assigned to a device until the client opens a device and that there is only one client per NDMPserver, whereas X11 is assigned to a display device before the first client connects and accepts connections from many clients.

RPC

The NDMP architecture is similar to RPC in that it uses XDR encoding. NDMP differs in that it is only defined for a TCP/IP connection and that it is not a call-return model, but rather a bi-directional asynchronous messaging model.

2.3 State Description

The following defines the DATA state diagram

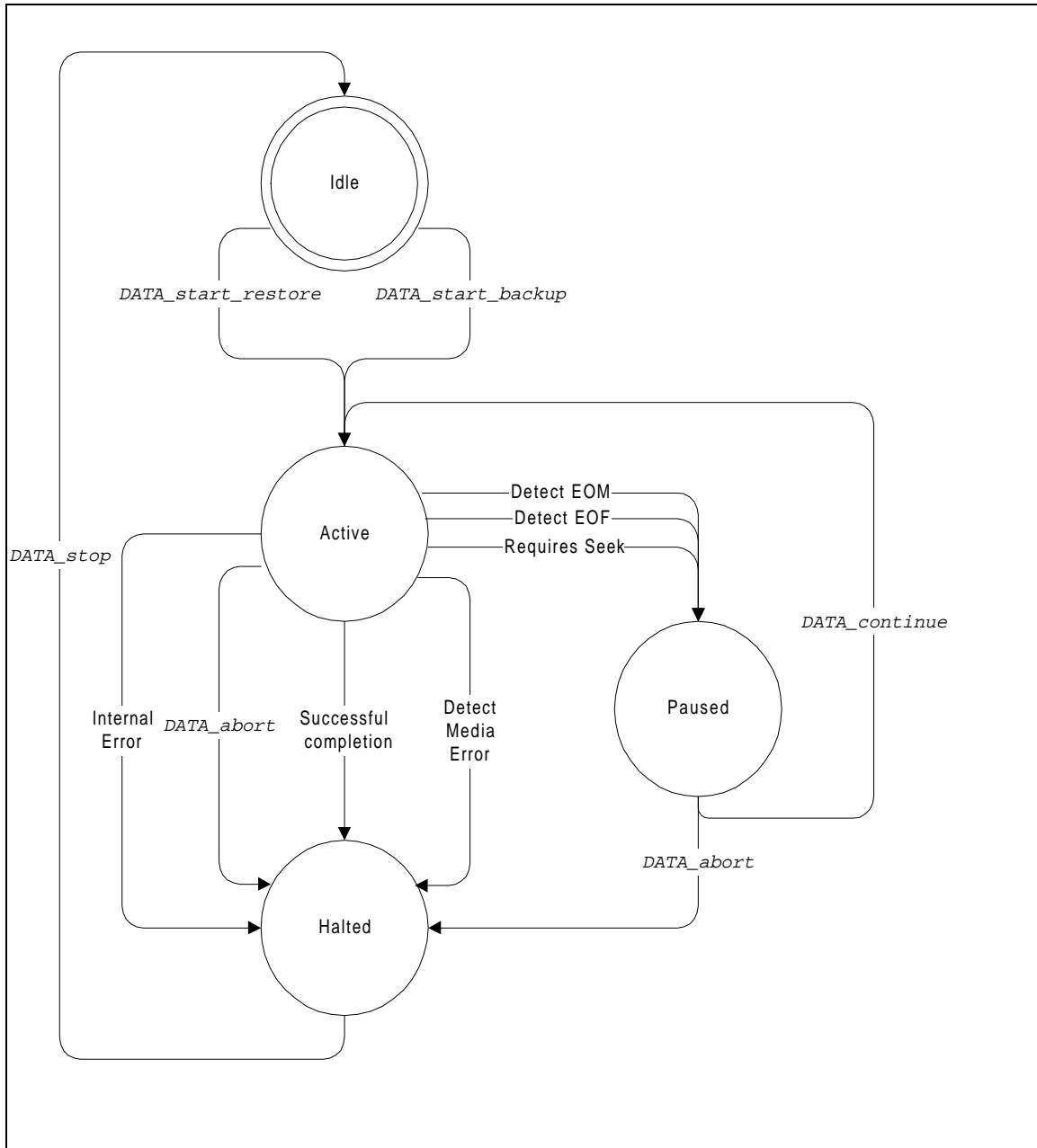


Figure 4 - Backup state diagram

2.4 INTERFACES

Messages are group together by functionality into several “interfaces”.

2.5 Messages From Backup Software To The NDMP Server

The NDMP server must implement the following interfaces

- CONNECT interface

This interface will be used when a client opens the communication to a NDMP server. This interface allows the NDMP server to authenticate the client and negotiate the version of protocol used.

- CONFIG interface

This interface allows backup software to discover the configuration of the NDMP server. It can be used to discover tape drives and jukeboxes as well as file systems and databases. backup software will use this interface to build backup software request and media server databases automatically.

- SCSI interface

This interface simply passes SCSI CDBs through to the SCSI device and returns the SCSI status. backup software will use this interface to control a locally attached jukebox. Software on the backup software host will construct SCSI CDBs and will interpret the returned status and data. This interface can also be used to exploit special features of SCSI tape drives.

- TAPE interface

The TAPE interface will support both tape positioning and tape read/write operations. backup software will use this interface to control the labeling and format of the tape. backup software will also use this interface for positioning of the tape during backups and restores.

- DATA interface

This is the interface that actually deals with the format of the backup data. backup software will initiate backups and restores using this interface. backup software provides all of the parameters that may affect the backup or restore using this interface. backup software does not place any constraints on the format of the backup data other than it must be a stream of data that can be written to the tape device.

2.6 Messages From NDMP Server To Backup Software

The NDMP server's implementation may send the following messages to the backup software host. All of the messages that the backup software host accepts are asynchronous. None of these messages will generate a reply message.

- NOTIFY interface

This message is used by the NDMP server to notify backup software that the NDMP server requires attention.

- FILE HISTORY interface

These messages allows the NDMP server to make entries in the file history for the current backup. The File History will be used by backup software to select files for retrieval.

- LOGGING interface

These messages allows the NDMP server to make entries in the backup log. This is used by the operator to monitor the progress and successful completion of the backup. It is also used to diagnose problems.

2.7 Messaging Protocol

The NDMP uses asynchronous messages and a message does not require a reply, however, many of the messages may result in a reply message.

2.8 Message Definitions

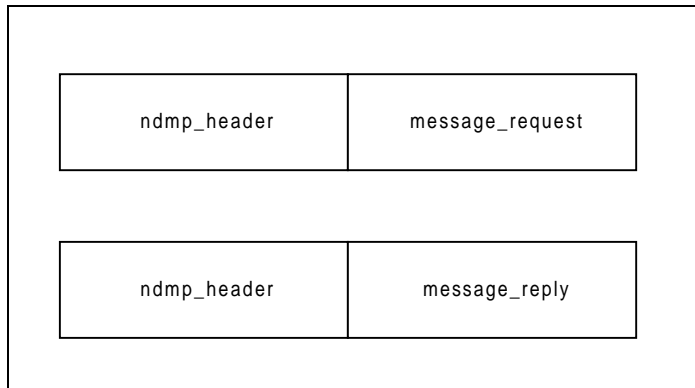
Each message is described using a block of XDR specification

```
/* message_name - */
/*      the message number is defined in enum ndmp_message */
struct message_name_request {
    int    dummy_request_argument;      /* Any arguments to request
message*/
};
struct message_name_reply {
    enum ndmp_error error;
    int    dummy_reply_argument;      /* Any arguments to reply message*/
};
```

The `ndmp_message` enum identifies the message. `Message_name` is a instance of `ndmp_message`. The body is a textual description of what the message is intended to do. If an error status is returned, then some of the reply arguments may be meaning less. If the request has no arguments, then the request structure is omitted. If the request does not expect a reply message, then the reply structure is omitted. `ndmp_error` enum identifies the error status.

2.9 Header

Each message will be preceded by a message header. The header will be used to identify the message, how to de-serialize the arguments and to dispatch the message.



The message headers are defined by the following XDR block

```

enum ndmp_header_message_type {
    NDMPMessageRequest, /* the request message */
    NDMPMessageReply, /* the reply message */
};
struct ndmp_header {
    u_long    sequence; /* monotonically increasing number */
    u_long    time_stamp; /* time stamp of message */
    enum ndmp_header_message_type message_type; /* what type of
message */
    enum ndmp_message    procedure; /* message number */
    u_long    reply_sequence; /* reply is in response to */
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotSupported: Unrecognized message */
    /* NDMPXDRDecodeErr: Can't decode the request message */
};
  
```

The sequence number is a number that starts at 1 and increases by 1 for every message sent. The client and the server both start with 1 and increase independently. The `time_stamp` is the julian time when the message was sent. The `ndmp_header_message_type` enum identifies whether the message is a request or a reply. If the message is a reply the `reply_sequence` is the sequence number from the request to which the reply belongs. Procedure identifies the message and is defined in each of the following XDR blocks. Valid procedure numbers are defined in section 3. If an error is returned, there will be no message_rply sent after the header message.

2.10 Error

The following errors are defined:

```
enum ndmp_error {
    NDMPNone, /* No error */
    NDMPNotSupported, /* Call is not supported*/
    NDMPDeviceBusy, /* The device is in use */
    NDMPDeviceOpened, /* Another tape or scsi device is already open*/
    NDMPNoAuth, /* Connection has not been authorized */
    NDMPPermission, /* Permission problem */
    NDMPNotOpened, /* SCSI device is not open */
    NDMPIOError, /* I/O error */
    NDMPTimeout, /* command timed out */
    NDMPIllegalArgs, /* illegal arguments in request */
    NDMPNoTapeLoaded, /* Can't open because there is no tape */
    NDMPWriteProtected, /* tape can't open for write */
    NDMPReachedEOF, /* Command encountered EOF */
    NDMPReachedEOM, /* Command encountered EOM */
    NDMPFileNotFound, /* File not found during restore */
    NDMPBadFile, /* The file descriptor is invalid */
    NDMPNoDevice, /* No device at the specified target */
    NDMPNoBus, /* invalid controller */
    NDMPXDRDecodeErr, /* Can't decode the request message */
    NDMPIllegalStat, /* Call can't be performed at this state */
    NDMPUndefined, /* Undefined error */
    NDMPXDREncodeErr, /* Can't encode the reply message */
    NDMPNoMEMErr /* No memory */
};
```

3. Interfaces

This section will define the interfaces. The defined interfaces and procedures are:

```
enum ndmp_message {
    CONNECT_OPEN=0x900,          /* CONNECT INTERFACE */
    CONNECT_AUTH,
    CONNECT_CLOSE,

    CONFIG_GET_HOST_INFO=0x100,   /* CONFIG INTERFACE */
    /* CONFIG_GET_DEVICES, */
    /* CONFIG_GET_DATA_SOURCES, */
    CONFIG_GET_BUTYPE_ATTR,

    SCSI_OPEN=0x200,             /* SCSI INTERFACE */
    SCSI_CLOSE,
    SCSI_GET_STATE,
    SCSI_SET_TARGET,
    SCSI_RESET_DEVICE,
    SCSI_RESET_BUS,
    SCSI_EXECUTE_CDB,

    TAPE_OPEN=0x300,             /* TAPE INTERFACE */
    TAPE_CLOSE,
    TAPE_GET_STATE,
    TAPE_MTIO,
    TAPE_WRITE,
    TAPE_READ,
    TAPE_SET_RECORD_SIZE,
    TAPE_EXECUTE_CDB,

    DATA_GET_STATE=0x400,        /* DATA INTERFACE */
    DATA_START_BACKUP,
    DATA_START_RECOVER,
    DATA_ABORT,
    DATA_GET_ENV,
    DATA_READ,
    DATA_RESVD1,
    DATA_STOP,
    DATA_CONTINUE,

    NOTIFY_PAUSED=0x500,         /* NOTIFY INTERFACE */
    NOTIFY_HALTED,
    NOTIFY_CONNECT,

    LOG_LOG=0x600,               /* LOGGING INTERFACE */
    LOG_DEBUG,
    LOG_FILE,

    FH_ADD_UNIX=0x700,           /* FILE HISTORY INTERFACE */
    FH_ADD_UNIX_DIR,
    FH_ADD_UNIX_NODE,
};
```

3.1 CONNECT Interface

This interface allows NDMP server to authenticate the client and negotiate the version of protocol which will be used.

3.1.1 Open Connection

```

/* CONNECT_OPEN */
struct connect_open_request {
    u_short ndmp_ver; /* the version of protocol supported */
};

struct connect_open_reply {
    enum ndmp_error error;
    /* NDMPNone: the version of protocol is supported */
    /* NDMPIllegalArgs: the version of protocol is not supported. */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
};

```

Used to negotiate the supported version of protocol. If the suggested version is not supported by NDMP, NDMPIllegalArgs is returned.

3.1.2 Authorization

```

/* CONNECT_AUTH */
enum ndmp_auth_type { /* the authentication method */
    NDMPAuthNone, /* no password is required */
    NDMPAuthText, /* the clear text password */
    NDMPAuthPCNFS /* pcnfs */
};

struct ndmp_auth_text {
    string user<>;
    string password<>;
};

struct ndmp_auth_pcnfs {
    string user<>;
    string password<>;
};

union ndmp_auth_data switch (enum ndmp_auth_type auth_type) {
    case NDMPAuthNone:
        void;
    case NDMPAuthText:
        struct ndmp_auth_text auth_text;
    case NDMPAuthPCNFS:
        struct ndmp_auth_pcnfs auth_pcnfs;
};

struct connect_auth_request {
    union ndmp_auth_data auth_data;
};

struct connect_auth_reply {
    enum ndmp_error error;
    /* NDMPNone: Authorization is granted */
    /* NDMPNoAuth: Authorization is denied */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
};

```

Used to authorize the NDMP connection. Some services may be denied if the connection is not successfully authorized.

3.1.3 Close Connection

```
/* CONNECT_CLOSE */
/* no request arguments */
/* no reply arguments */
```

This message is used when client wants to close the NDMP connection.

3.2 CONFIG Interface

This interface allows backup software to discover the configuration of the NDMP server.

3.2.1 Get Host Info

```
/* CONFIG_GET_HOST_INFO */
/* no request arguments */

struct config_get_host_info_reply {
    enum ndmp_error error;
    /* NDMPNone : no error */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */

    string      name<>; /* host name */
    string      os_type<>; /* The operating system type (i.e.
SOLARIS) */
    string      os_vers<>; /* The version number of the OS (i.e. 2.5)
*/
    string      hostid<>; /* host identifier (i.e. 12345678) */
    enum ndmp_auth_type auth_type<>;
};
```

auth_type returns a list of the authentication methods which NDMP server supports.

3.2.2 Get backup Type Attribute

```
/* CONFIG_GET_BUTYPE_ATTR */
const NDMP_NO_BACKUP_FILELIST=      0x0001;
const NDMP_NO_BACKUP_FHINFO=        0x0002;
const NDMP_NO_RECOVER_FILELIST=      0x0004;
const NDMP_NO_RECOVER_FHINFO=        0x0008;
const NDMP_NO_RECOVER_SSID=          0x0010;
const NDMP_NO_RECOVER_INC_ONLY=      0x0020;

struct config_get_butype_attr_request {
    string name<>; /* backup type name */
};
struct config_get_butype_attr_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPIllegalArgs: specified backup type is not supported */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
    u_long      bu_attr;
```

This message is used to query the capability of the supported backup type. The following attributes are defined:

NDMP_NO_BACKUP_FILELIST - NDMP server doesn't support archiving of selective files as specified by a file list. (i.e. only supports dumping the entire file system.)

NDMP_NO_BACKUP_FILEINFO - NDMP server doesn't support the file history.

NDMP_NO_RECOVER_FILELIST - NDMP server doesn't support restoration of individual files.

NDMP_NO_RECOVER_FHINFO - NDMP server doesn't support the direct access restore (i.e. positioning to the offset of a backup image and restore the specified file).

NDMP_NO_RECOVER_SSID - NDMP server doesn't support the true image retrieval (i.e. deleted files will be restored also).

NDMP_NO_RECOVER_INC_ONLY - NDMP server doesn't support incremental only restoration (i.e. a full restore must be performed prior to an incremental restore).

3.3 SCSI Interface

The SCSI interface allows low level control of SCSI devices such as jukeboxes.

3.3.1 Open SCSI Device

```
/* SCSI_OPEN */
struct scsi_device {
    string device_name<>;
};
struct scsi_open_request {
    struct scsi_device    device;
};
struct scsi_open_reply {
    enum ndmp_error error;
    /* NDMPNone:           The device is open successfully */
    /* NDMPDeviceBusy:     The device is in use */
    /* NDMPDeviceOpened:  Another tape or scsi device is already open
*/
    /* NDMPNotSupported:  not supported for this implementation */
    /* NDMPIllegalStat:   call can't be performed at this state. */
};
```

Opens the SCSI device. This operation is required before any other SCSI requests can be executed. The open must be an exclusive open. Only one NDMP server can open a SCSI device at a time. The NDMP server can only open one SCSI or tape device at a time. A NDMPDeviceBusy is returned if the NDMP server already has a tape or SCSI device opened.

3.3.2 Close Device

```
/* SCSI_CLOSE */
/* no request arguments */
struct scsi_close_reply {
    enum ndmp_error error;
    /* NDMPNone:           close was successful */
    /* NDMPNotOpened:     SCSI device is not open */
    /* NDMPNotSupported:  not supported for this implementation */
    /* NDMPIllegalStat:   call can't be performed at this state. */
};
```


No further requests can be until another open request is successfully executed.

3.3.3 Get SCSI State

```
/* SCSI_GET_STATE */
/* no request arguments */
struct scsi_get_state_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: SCSI device is not open */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */

    short target_controller;
    short target_id;
    short target_lun;
};
```

Return the current state of the SCSI interface. The target information provides information about which SCSI device is controlled by this interface.

3.3.4 Set SCSI Target

```
/* SCSI_SET_TARGET */
struct scsi_set_target_request {
    struct scsi_device device;
    u_short target_controller;
    u_short target_id;
    u_short target_lun;
};
struct scsi_set_target_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNoBus: Invalid controller */
    /* NDMPNoDevice: No device at this specified target */
    /* NDMPUndefined: other error */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
};
```

Selects or changes the SCSI target. When the SCSI interface is opened, we do not know if the NDMP server has opened a device file that can pass commands to a single SCSI target or to multiple SCSI targets. This part of protocol allows us to pass the information describing the SCSI target to which to send commands. Additionally, if the target can talk to multiple targets, this allows us to “scan” the SCSI bus on the NDMP host for diagnostics or the jukebox discovery.

3.3.5 Reset Device

```
/* SCSI_RESET_DEVICE */
/* no request arguments */
struct scsi_reset_device_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: SCSI device is not opened */
    /* NDMPNotSupported: Not supported by this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */};
```

Send a SCSI device reset message to the SCSI device.

3.3.6 Reset Bus

```
/* SCSI_RESET_BUS */
/* no request arguments */
struct scsi_reset_bus_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: SCSI device is not opened */
    /* NDMPNotSupported: Not supported by this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */};
```

Assert a SCSI bus reset on the SCSI bus to which the SCSI device is attached.

3.3.7 Execute CDB

```
/* SCSI_EXECUTE_CDB */
const DATA_IN = 0x00000001; /* Expect data from SCSI device */
const DATA_OUT = 0x00000002; /* Transfer data to SCSI device */

struct scsi_execute_cdb_request {
    u_long flags;
    u_long timeout;
    u_long datain_len; /* set for expected datain */
    opaque cdb<>;
    opaque dataout<>;
};
struct scsi_execute_cdb_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: SCSI device is not opened */
    /* NDMPNotSupported: Not supported by this implementation */
    /* NDMPPIOError: I/O error */
    /* NDMPTimeout: command timed out */
    /* NDMPIllegalArgs: illegal arguments in request */
    /* NDMPBadFile: Bad file descriptor */
    /* NDMPIllegalStat: call can't be performed at this state. */
};
u_char status; /* SCSI status byte */
u_long dataout_len; /* set for the expected dataout */
opaque datain<>; /* SCSI datain */
opaque ext_sense<>; /* Extended sense data */
};
```

Send a SCSI Control Data Block to a SCSI device. If a check condition is generated, then the extended sense data is also retrieved. Data can be transferred to or from the SCSI device as part of the command.

The server selects the SCSI target. The cdb is sent to the SCSI device in command mode. If DATA_OUT is

set in the flags, then the `data_out` is sent to the SCSI device in data mode. Sometimes, the host will disconnect from the target and waits for a re-select. If `timeout` is zero then the host will wait indefinitely for the target to re-select. If `timeout` is non-zero then the host will wait `timeout` milliseconds for the target to reselect. If the reselect does not occur then an `NDMPTimeout` exception occurs. If the target re-selects and the status is `CHECK CONDITION`, then the server executes a `REQUEST SENSE` cdb. If the `DATA_IN` flag is set, then server gets data from the target in a data mode and `datain_len` and `dataout_len` are set to indicate the expected data length. The SCSI status, the data in and the extended sense data is returned.

`DATA_IN` and `DATA_OUT` are with reference to the host. They refer to data from the SCSI device into the host and data out of the host and to the SCSI device.

3.4 TAPE Interface

Provide complete control of a tape drive. If the tape drive is a SCSI tape drive, then this interface also provides low level CDB access to the tape drive. This interface is analogous to the `rmt` protocol. The physical device is assigned when the server is started.

3.4.1 Open Tape Device

```
/* TAPE_OPEN */
struct tape_device {
    string device_name<>;
};
enum tape_open_rwmode {
    NDMPread_only,
    NDMPread_write
};
struct tape_open_request {
    struct tape_device device_name;
    enum tape_open_rwmode rwmode;
};
struct tape_open_reply {
    enum ndmp_error error;
    /* NDMPNone:          The tape drive is open successfully */
    /* NDMPNoTapeLoaded: Cannot open because there is no tape */
    /* NDMPWriteProtected: tape cannot be open for write */
    /* NDMPDeviceBusy:   Another process has the device open */
    /* NDMPDeviceOpened: Another tape or scsi device is open */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat:  call can't be performed at this state. */
};
```

Opens the tape device in the desired mode. This operation is required before any other requests can be executed. The device is opened exclusively, no other NDMP server can open the device. If a tape is not in the drive then a `NDMPNoTapeLoaded` error is returned. If the media is write protected, then the device cannot be opened in `NDMPread_write` mode and a `NDMPWriteProtected` error is returned. A `NDMPDeviceOpened` is returned if the NDMP server already has a tape or SCSI device opened.

3.4.2 Close Device

```

/* TAPE_CLOSE */
/* no request arguments */
struct tape_close_reply {
    enum ndmp_error error;
    /* NDMPNone:          close was successful */
    /* NDMPNotOpened:    tape device is not open */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat:  call can't be performed at this state. */
};

```

Close the tape drive. Drive can be opened by others. No further requests can be processed until another open request is successfully executed.

3.4.3 Get Tape State

```

/* TAPE_GET_STATE */
/* no request arguments */
/* the flag for tape_stat */
const NDMPNOREWIND= 0x0008; /* non-rewind device */
const NDMPWRITEPROTECT= 0x0010; /* write-protected */
const NDMPMEDIAERROR= 0x0020; /* Media error */
const NDMPUNLOAD= 0x0040; /* tape will be unloaded when device is
closed */
struct tape_get_state_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: the device is not opened */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
    u_long    tape_stat;
    u_long    filenum;
    u_long    recno;
    u_long    record_size;
    int    softerror; /* soft error */
    u_long    block_size; /* block size */
    u_long    blockno; /* block number */
    u_quad    total_space; /* total space on volume */
    u_quad    space_remain; /* space remaining */
};

```

Return the state of the tape drive interface. Filenum, recno and blockno all start with 0 and represent the next tape record to be read or written. NDMPMEDIAERROR is not cleared until the next tape operation is performed. The total_space is the kilobytes of the current tape. The value will be 0 if not supported. The space_remain is the space available (kilobytes) on the tape. The value will be 0 if not supported.

3.4.4 MTIO

```

/* TAPE_MTIO */
enum tape_mtio_op {
    NDMPfsf,
    NDMPbsf,
    NDMPfsr,
    NDMPbsr,
    NDMPrew,
    NDMPweof,
    NDMPoff
};

struct tape_mtio_request {
    enum tape_mtio_op tape_op;
    u_long          count;
};

struct tape_mtio_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: tap is not opened */
    /* NDMPIllegalArgs: illegal arguments */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
    u_long          resid_count;
};

```

Provide access to the standard mtio operations.

NDMPfsf	Forward space over file marks
NDMPbsf	Backward space over file marks
NDMPfsr	Forward space over tape records
NDMPbsr	Backward space over tape records
NDMPrew	Rewind the tape
NDMPweof	Write end-of-file marks
NDMPoff	Eject the tape from the drive

When spacing forward over a record, the tape head is positioned in the tape gap between the record just skipped and the next record. When spacing forward over file marks, the tape head is positioned in the tape gap between the next file mark and the record that follows it. When spacing backward over a record data, the tape head is positioned in the tape gap immediately preceding the tape record where the tape head is currently positioned. When spacing backward over file marks, the tape head is positioned in the tape gap preceding the file mark the next read would fetch the EOF. `Resid_count` returns the number of files or records unprocessed.

3.4.5 Write

```

/* TAPE_WRITE */
struct  tape_write_request {
    opaque      data_out<>;
};
struct  tape_write_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: tape device is not open */
    /* NDMPIOError: I/O error */
    /* NDMPReached_EOM: Reach end of medium */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
    u_long      count;
};

```

Write the data to the tape drive. Count returns the number of bytes of data written.

3.4.6 Read

```

/* TAPE_READ */
struct  tape_read_request {
    u_long      count;
};
struct  tape_read_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotOpened: tape device is not open */
    /* NDMPIOError: I/O error */
    /* NDMPReached_EOF: Reach end of file */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
    opaque      data_in<>;
};

```

Read data from the tape drive. If an media error is encountered then a NDMPIOError error is returned. If the read encounters a file mark, then a NDMPReached_EOF error is returned.

3.4.7 Set Record Size

```

/* TAPE_SET_RECORD_SIZE */
struct tape_set_record_size_request {
    u_long len;
};
struct tape_set_record_size_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPIllegalArgs: len exceeds the implementation limit */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
}

```

Set the tape record size. Every write to the tape drive should be this size. The data interface should buffer the data to this size for I/O. If the specified len exceeds the NDMP limitation, NDMPIllegalArgs is returned.

3.4.8 Execute CDB

```

/* TAPE_EXECUTE_CDB */
typedef scsi_execute_cdb_request tape_execute_cdb_request;
typedef scsi_execute_cdb_reply tape_execute_cdb_reply;

```

This message behaves in exactly the same way as the SCSI_EXECUTE_CDB except that it sends the cdb to the tape device. This request can't be used to change the state of the tape device.

3.5 DATA Interface

Selects and formats data for backup. Extracts files from backup for retrieval.

3.5.1 Get Data State

```

/* DATA_GET_STATE */
/* no request arguments */
enum data_operation {
    NDMPNoaction,
    NDMPbackup,
    NDMPRestore,
};
enum data_state {
    NDMPIdle,
    NDMPActive,
    NDMPPaused,
    NDMPHalted,
};
enum data_halt_reason {
    NDMPNA_HALT,
    NDMPSuccessful,
    NDMPAborted,
    NDMPMediaError,
    NDMPInternalError,
    NDMPNoSplit,
};
enum data_pause_reason {

```

```

        NDMPNA_PAUSE,
        NDMPEOM,
        NDMPEOF,
        NDMPRESVD,
    };

    struct data_get_state_reply {
        enum ndmp_error error;
        /* NDMPNone: no error */
        /* NDMPNotSupported: not supported for this implementation */
        /* NDMPIllegalStat: call can't be performed at this state. */

        enum data_operation data_operation;
        enum data_state data_state;
        enum data_halt_reason data_halt_reason;
        enum data_pause_reason data_pause_reason;
        struct u_quad resvd1;
        struct u_quad bytes_processed;
        struct u_quad est_bytes_remain;
        u_long est_time_remain;
        struct u_quad resvd2;
        struct u_quad resvd3;
    };

```

The state is applicable to both backup and recover processes. `bytes_processed` is the number of bytes of data backed up or the number of bytes of data read for a restore. `est_bytes_remain` is the number of bytes of backup data left to read or write. The `est_time_remain` is an estimate of the number of seconds before the backup or restore will be complete. The `est_time_remain` and `est_bytes_remain` should be best guesses but there is no accuracy requirement placed these values. A value of zero should be used for values that cannot be estimated.

3.5.2 Backup

```

/* DATA_START_BACKUP */
struct pval {
    string name<>;
    string value<>;
};

struct data_start_backup_request {
    string bu_type<>; /* backup method to use */
    struct pval env<>; /* Parameters that may modify
    backup */
};

struct data_start_backup_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPIllegalArgs: wrong argument is given */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
};

```

Begins a backup. The `id` identifies the object to be backed up. The meaning of `id` is dependent. The type of backup is also implementation dependent. The `env` is a list of parameters that may affect the behavior of the backup. The `env` returned by the `DATA_GET_ENV` will be saved and made available to the retrieval process.

The following environmental variables are defined by backup software.

Variable Name	Meaning	Value
PREFIX	the prefix path for this request	path name
TYPE	the backup software backup class type	the value could be different from the bu_type passed to NDMP server.
USER	user id to perform backup	user name
HIST	a flag to maintain file history	y/n
ID	backup software request identifier	the value is the same as the id passed to NDMP server.
HOST	host name	host name
LEVEL	dump level	0 - 9

Backup software allows user to define their own variable names in the configuration file. For example:

The following environmental variables are defined by backup software dump backup class.

Variable Name	Meaning	Value
filesystem	device or file system name to be backed up	file system or device name (e.g. /dev/rsd0a)

The following environmental variables are defined by backup software tar backup class.

Variable Name	Meaning	Value
files	a list of files to be backed up	e.g. /* /*.c /*h

The following environmental variables are defined by backup software cpio backup class.

Variable Name	Meaning	Value
cmd	the command which is used to generate the file list for cpio.	e.g. find . -print

3.5.3 Recover

```

/* DATA_START_RECOVER */
struct name {
    string      name<>;
    string      dest<>;
    u_short     ssid;
    u_quad      fh_info;
};
struct data_start_recover_request {
    struct      pval  env<>;
    struct name nlist<>;
    string      bu_type<>;
};
struct data_start_recover_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPIllegalArgs: wrong argument is given */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
};

```

Recover the files specified in `nlist` from the backup. The `env` is the list of parameters and values saved at the end of the backup.

The name is the original relative path name to the backup root directory. Dest is the full destination path name for the file. Ssid is the selection set id. Ssid is used when restoring a directory from an incremental backup. Ssid starts with 1. The value of ssid will be 0 if not supported. The directory will be restored from the incremental and full backup using the same ssid. Fh_info is the data save by the fh_unix message during backup. Fh_info is to be used by the restore program to perform tape positioning commands for fast data retrieval and typically contains a byte or record offset from the beginning of the backup data.

3.5.4 Abort

```
/* DATA_ABORT */
/* no request arguments */

struct data_abort_reply {
    ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPUndefined: Unknown error */
};
```

Send a message to abort the current backup or restore. The operation should be terminated as soon as possible.

3.5.5 Stop

```
/* DATA_STOP */
/* no request arguments */

struct data_stop_reply {
    ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPUndefined: Unknown error */
};
```

Send a message to inform NDMP server that the current backup is complete. NDMP server will change to idle state and be ready for process the other request.

3.5.6 Continue

```
/* DATA_CONTINUE */
/* no request arguments */

struct data_continue_reply {
    ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPUndefined: Unknown error */
};
```

Send a message to resume the backup due to EOM.

3.5.7 Get ENV

```
/* DATA_GET_ENV */
/* no request arguments */
struct data_get_env_reply {
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPNotSupported: not supported for this implementation */
    /* NDMPIllegalStat: call can't be performed at this state. */
    struct pval env<>;
};
```

Get the backup environment. This began with the environment included in the DATA_START_BACKUP message but may be modified by the backup process. The environment will be made available to the restore process.

4. Backup Software Interfaces

These interfaces are available to the NDMP server. Only the requests that are intended for use by the NDMP server are available. Many of the interfaces require initialization which will be performed by backup software before the NDMP server is called.

4.1 NOTIFY Interface

This interface is used by the NDMP server to let backup software know that the NDMP server requires attention.

4.1.1 Notify Paused

```
/* NOTIFY_PAUSED */
struct notify_paused_request {
    enum data_pause_reason reason;
    struct u_quad resvd;
};
/* No reply */
```

This message is used to notify backup software that the NDMP server has paused.

4.1.2 Notify Halted

```
/* NOTIFY_HALTED */
struct notify_halted_request {
    enum data_halt_reason reason;
    string text_reason<>;
};
/* No reply */
```

This message is used to notify backup software that the NDMP server has halted. The text_reason string is used for diagnostic purposes only.

4.1.3 Notify Connect

```
/* NOTIFY_CONNECT */
enum connect_reason {
    NDMPConnected,      /* Connect successfully */
    NDMPHostDown,       /* Connection shutdown */
    NDMPConnRefused     /* Reach the maximum number of connections */
};
struct notify_connect_request {
    enum connect_reason reason;
    u_short ndmp_ver;   /* the latest NDMP supported */
    string text_reason<>;
};
/* No reply */
```

This message is used to notify backup software that the NDMP server is connected successfully or not. The text_reason string is used for diagnostic purposes only.

4.2 LOGGING Interface

Make an entry into a backup specific log.

4.2.1 Log

```
/* LOG_LOG */
struct log_log_request {
    string      entry<>;
};
/* No reply */
```

Make an entry into the log file for the backup.

4.2.2 Debug

```
/* LOG_DEBUG */
enum debug_level {
    NDMPuser_info,
    NDMPuser_summary,
    NDMPuser_detail,
    NDMPdiag_info,
    NDMPdiag_summary,
    NDMPdiag_detail,
    NDMPprog_info,
    NDMPprog_summary,
    NDMPprog_detail
};
struct log_debug_request {
    enum debug_level level;
    string message<>;
};
/* No reply */
```

Make an entry in the diagnostic log file. The level is divided into two components. The first component is the intended audience. The audience can be the end user (user), the technical support personnel (diag), or the development engineer (prog). The second component is the level of detail requested. The level of detail is specified as info, summary and detail. There are no specific guidelines on the use of level of detail, but a message that typically is encountered less than 10 times during a backup should be an info level. While a message that is encountered more than 100 times should be at a detail level.

4.2.3 File Recovered

```
/* LOG_FILE */
struct log_file_request {
    string      name<>;
    u_short     ssid;
    enum ndmp_error error;
    /* NDMPNone: no error */
    /* NDMPPermission: permission denied */
    /* NDMPFileNotFound: File not found during restore */
};
/* No reply */
```

This message is used to inform backup software that a file on the file recovery list has or has not been recovered. If the file is successfully recovered then the error status is NDMPNone. If the file cannot be recovered because it is missing on the backup, then the status is NDMPFileNotFound, and if the file cannot be recovered due to permission problems, then the error is NDMPPermission. This log message allows backup software to monitor the progress of a recovery.

4.3 FILE HISTORY Interface

backup software maintains a file by file record of every file that is backed up. These record are called the file history. This interface provides the ability to add entries to the file history. backup software currently only maintains file history in a format that is compatible with a UNIX file system, however the interface is defined in an extensible manner.

4.3.1 Add Unix Path

```
/* FH_ADD_UNIX */
typedef string unix_path<>;
enum unix_file_type {
    NDMPfsdir,
    NDMPfsfifo,
    NDMPfscspec,
    NDMPfsbspec,
    NDMPfsregf,
    NDMPfsslink,
    NDMPfssock
};
struct unix_file_stat {
    unix_file_type    ftype;
    u_long            mtime;
    u_long            atime;
    u_long            ctime;
    u_long            uid;
    u_long            gid;
    u_long            mode;
    u_quad            size;
    u_quad            fh_info;
}
struct unix_fh_entry {
    unix_path name;
    struct unix_file_stat file_stat;
};

struct fh_add_unix_request {
    struct unix_fh_entry unix_fh_entries<>;
};
/* No reply */
```

Adds a sequence of unix file history entries to the file history database.

4.3.2 Add Unix Dir

```
/* FH_ADD_UNIX_DIR */
struct unix_dir_ent {
    unix_path    name;
    u_long       node;
    u_long       parent;
}
struct fh_add_unix_dir_request {
    struct unix_dir_ent dir<>;
};
/* No reply */
```

This message is used to support directory/inode types of backup formats. The `node` number can be any unique number that matches a corresponding `fh_add_unix_node` message. `parent` is the parent directory inode number. `name` is the file/directory name relative to the current directory, e.g. `testdir`.

4.3.3 Add Unix Node

```
/* FH_ADD_UNIX_NODE */
struct unix_node {
    struct        unix_file_stat fstat;
    u_long       node;
}
struct fh_add_unix_node_request {
    struct        unix_node nlist<>;
};
/* No reply */
```

Add a list of attribute information to a directory/inode type of file history. These entries must match a corresponding node number in a add directory message. For each file, this message must be sent after the corresponding `fh_add_unix_dir` message.

5. Appendix A - ndmp.x protocol file

```

const NDMPVER = 1;

struct      u_quad {
    u_long      high;
    u_long      low;
};
struct pval {
    string      name<>;
    string      value<>;
};
struct scsi_device {
    string device_name<>;
};
struct tape_device {
    string device_name<>;
};

enum ndmp_error {
    NDMPNone, /* No error */
    NDMPNotSupported, /* Call is not supported */
    NDMPDeviceBusy, /* The device is in use */
    NDMPDeviceOpened, /* Another tape or scsi device */
    /* is already open */
    NDMPNoAuth, /* connection has not been authorized */
    NDMPPermission, /* some sort of permission problem */
    NDMPNotOpened, /* SCSI device is not open */
    NDMPIOError, /* I/O error */
    NDMPTimeout, /* command timed out */
    NDMPIllegalArgs, /* illegal arguments in request */
    NDMPNoTapeLoaded, /* Cannot open because there is */
    NDMPWriteProtected, /* tape cannot be open for write */
    NDMPReachedEOF, /* Command encountered EOF */
    NDMPReachedEOM, /* Command encountered EOM */
    NDMPFileNotFound, /* File not found during restore */
    NDMPBadFile, /* The file descriptor is invalid */
    NDMPNoDevice, /* The device is not at that target */
    NDMPNoBus, /* Invalid controller */
    NDMPXDRDecodeErr, /* Can't decode the request argument */
    NDMPIllegalStat, /* Call can't be performed at this state */
    /*
    NDMPUndefined, /* Undefined Error */
    NDMPXDREncodeErr, /* Can't encode the reply argument */
    NDMPNoMEMErr /* no memory */
    */
};

enum ndmp_header_message_type {
    NDMPMessageRequest,
    NDMPMessageReply
};

enum ndmp_message {
    CONNECT_OPEN = 0x900, /* CONNECT INTERFACE */
    CONNECT_AUTH,
    CONNECT_CLOSE,

    CONFIG_GET_HOST_INFO=0x100, /* CONFIG INTERFACE */
    CONFIG_AUTH,
    CONFIG_GET_BUTYPE_ATTR,

    SCSI_OPEN=0x200, /* SCSI INTERFACE */
    SCSI_CLOSE,

```

```

SCSI_GET_STATE,
SCSI_SET_TARGET,
SCSI_RESET_DEVICE,
SCSI_RESET_BUS,
SCSI_EXECUTE_CDB,

TAPE_OPEN=0x300,          /* TAPE INTERFACE */
TAPE_CLOSE,
TAPE_GET_STATE,
TAPE_MTIO,
TAPE_WRITE,
TAPE_READ,
TAPE_SET_RECORD_SIZE,
TAPE_EXECUTE_CDB,

DATA_GET_STATE=0x400,      /* DATA INTERFACE */
DATA_START_BACKUP,
DATA_START_RECOVER,
DATA_ABORT,
DATA_GET_ENV,
DATA_READ,
DATA_RESVD1,
DATA_STOP,
DATA_CONTINUE,

NOTIFY_PAUSED=0x500,       /* NOTIFY INTERFACE */
NOTIFY_HALTED,
NOTIFY_CONNECT,

LOG_LOG=0x600,             /* LOGGING INTERFACE */
LOG_DEBUG,
LOG_FILE,

FH_ADD_UNIX=0x700,         /* FILE HISTORY INTERFACE */
FH_ADD_UNIX_DIR,
FH_ADD_UNIX_NODE,

};

struct ndmp_header {
    u_long    sequence;      /* monotonically increasing number
*/
    u_long    time_stamp;    /* time stamp of message */
    ndmp_header_message_type message_type; /* what type of message */
    enum ndmp_message_procedure; /* message number */
    u_long    reply_sequence; /* reply is in response to */
    ndmp_error error;        /* communications errors */
};

```

CONNECT INTERFACE

```

/* CONNECT_OPEN */
struct connect_open_request {
    u_short ndmp_ver; /* the version of protocol supported */
};
struct connect_open_reply {
    ndmp_error error;
};
/* CONNECT_AUTH */
enum ndmp_auth_type {
    NDMPAuthNone,    /* no password is required */
    NDMPAuthText,    /* the clear text password */
    NDMPAuthPCNFS    /* pcnfs */
};

```

```

};
struct ndmp_auth_text {
    string user <>;
    string password<>;
};
struct ndmp_auth_pcnfs {
    string user <>;
    string password <>;
};
union ndmp_auth_data switch (enum ndmp_auth_type auth_type) {
    case NDMPAuthNone:
        void;
    case NDMPAuthText:
        struct ndmp_auth_text    auth_text;
    case NDMPAuthPCNFS:
        struct ndmp_auth_pcnfs   auth_pcnfs;
};
struct connect_auth_request {
    ndmp_auth_data    auth_data;
};
struct connect_auth_reply {
    ndmp_error error;
};

/* CONNECT_CLOSE */
/* no request arguments */
/* no reply arguments */

```

CONFIG INTERFACE

```

/* CONFIG_GET_HOST_INFO */
/* no request arguments */
struct config_get_host_info_reply {
    ndmp_error error;
    string      name<>;                /* host name */
    string      os_type<>;             /* The operating system type (i.e.
SOLARIS) */
    string      os_vers<>;             /* The version number of the OS
(i.e. 2.5) */
    string      hostid<>;
    ndmp_auth_type    auth_type<>;
};

/* CONFIG_GET_BUTYPE_ATTR */
const NDMP_NO_BACKUP_FILELIST =    0x0001;
const NDMP_NO_BACKUP_FHINFO =     0x0002;
const NDMP_NO_RECOVER_FILELIST =   0x0004;
const NDMP_NO_RECOVER_FHINFO =     0x0008;
const NDMP_NO_RECOVER_SSID =       0x0010;
const NDMP_NO_RECOVER_INC_ONLY =   0x0020;

struct config_get_butype_attr_request {
    string name <>;    /* backup type name */
};

struct config_get_butype_attr_reply {
    ndmp_error error;
    u_long      bu_attr;
};

```

SCSI INTERFACE

```

/* SCSI_OPEN */
struct scsi_open_request {
    scsi_device device;
};

```

```

struct scsi_open_reply {
    ndmp_error error;
};

/* SCSI_CLOSE */
/* no request arguments */
struct scsi_close_reply {
    ndmp_error error;
};

/* SCSI_GET_STATE */
/* no request arguments */
struct scsi_get_state_reply {
    ndmp_error error;
    short target_controller;
    short target_id;
    short target_lun;
};

/* SCSI_SET_TARGET */
struct scsi_set_target_request {
    scsi_device device;
    u_short target_controller;
    u_short target_id;
    u_short target_lun;
};

struct scsi_set_target_reply {
    ndmp_error error;
};

/* SCSI_RESET_DEVICE */
/* no request arguments */
struct scsi_reset_device_reply {
    ndmp_error error;
};

/* SCSI_RESET_BUS */
/* no request arguments */
struct scsi_reset_bus_reply {
    ndmp_error error;
};

/* SCSI_EXECUTE_CDB */
const DATA_IN= 0x00000001; /* Expect data from SCSI device */
const DATA_OUT= 0x00000002; /* Transfer data to SCSI device */

struct scsi_execute_cdb_request {
    u_long flags;
    u_long timeout;
    u_long datain_len; /* Set for expected datain */
    opaque cdb<>;
    opaque dataout<>;
};

struct scsi_execute_cdb_reply {
    ndmp_error error;
    u_char scsi_status; /* SCSI status bytes */
    u_long dataout_len;
    opaque datain<>; /* SCSI datain */
    opaque ext_sense<>; /* Extended sense data */
};

```

TAPE INTERFACE

```

/* TAPE_OPEN */
enum tape_open_rwmode {
    NDMPread_only,
    NDMPread_write
};
struct tape_open_request {
    tape_device device_name;
    tape_open_rwmode rwmode;
};
struct tape_open_reply {
    ndmp_error error;
};

/* TAPE_CLOSE */
/* no request arguments */
struct tape_close_reply {
    ndmp_error error;
};

/*TAPE_GET_STATE */
/* no request arguments */
const NDMPTPNOREWIND = 0x0008;    /* non-rewind device */
const NDMPTPPROTECT = 0x0010;    /* write-protected */
const NDMPTPERROR = 0x0020;    /* media error */
const NDMPUNLOAD = 0x0040;    /* tape will be unloaded when device
is closed */

struct tape_get_state_reply {
    ndmp_error error;
    u_long tape_stat;
    u_long filenum;
    u_long recno;
    u_long record_size;
    int softerror;
    u_long block_size;
    u_long blockno;
    u_quad total_space;
    u_quad space_remain;
};

/* TAPE_MTIO */
enum tape_mtio_op {
    NDMPfsf,
    NDMPbsf,
    NDMPfsr,
    NDMPbsr,
    NDMPrew,
    NDMPweof,
    NDMPoff
};
struct tape_mtio_request {
    enum tape_mtio_op tape_op;
    u_long count;
};
struct tape_mtio_reply {
    ndmp_error error;
    u_long resid_count;
};

/* TAPE_WRITE */

```

```

struct      tape_write_request {
    opaque      data_out<>;
};
struct      tape_write_reply {
    ndmp_error error;
    u_long      count;
};

/* TAPE_READ */
struct      tape_read_request {
    u_long      count;
};
struct      tape_read_reply {
    ndmp_error error;
    opaque      data_in<>;
};

/* TAPE_SET_RECORD_SIZE */
struct      tape_set_record_size_request {
    u_long      len;
};
struct      tape_set_record_size_reply {
    ndmp_error error;
};

/* TAPE_EXECUTE_CDB */
typedef scsi_execute_cdb_request      tape_execute_cdb_request;
typedef scsi_execute_cdb_reply       tape_execute_cdb_reply;

```

DATA INTERFACE

```

/* GET_DATA_STAE */
/* no request arguments */
enum data_operation {
    NDMPNoaction,
    NDMPbackup,
    NDMPRestore
};
enum data_state {
    NDMPIdle,
    NDMPActive,
    NDMPPaused,
    NDMPHalted
};
enum data_halt_reason {
    NDMPNA_HALT,
    NDMPSuccessful,
    NDMPAborted,
    NDMPMediaError,
    NDMPInternalError,
    NDMPNoSplit
};
enum data_pause_reason {
    NDMPNA_PAUSE,
    NDMPPEOM,
    NDMPPEOF,
    NDMPRESVD
};
struct data_get_state_reply {
    ndmp_error error;
    data_operation data_operation;
    data_state data_state;
    data_halt_reason data_halt_reason;
};

```

```

        data_pause_reason data_pause_reason;
        u_quad          reserved1;
        u_quad          bytes_processed;
        u_quad          est_bytes_remain;
        u_long          est_time_remain;
        u_quad          reserved2;
        u_quad          reserved3;
};

/* DATA_START_BACKUP */
struct data_start_backup_request {
    string      bu_type<>;
    pval  env<>;
    /* backup method to use */
    /* Parameters that may modify backup */
};
struct data_start_backup_reply {
    ndmp_error error;
};

/* DATA_START_RECOVER */
struct name {
    string      name<>;
    string      dest<>;
    u_short     ssid;
    u_quad      fh_info;
};
struct data_start_recover_request {
    pval  env<>;
    name  nlist<>;
    string      bu_type<>;
};
struct data_start_recover_reply {
    ndmp_error error;
};

/* DATA_ABORT */
/* no request arguments */
struct data_abort_reply {
    ndmp_error error;
};

/* DATA_STOP */
/* no request arguments */
struct data_stop_reply {
    ndmp_error error;
};

/* DATA_CONTINUE */
/* no request arguments */
struct data_continue_reply {
    ndmp_error error;
};

/* DATA_GET_ENV */
/* no request arguments */
struct data_get_env_reply {
    ndmp_error error;
    pval  env<>;
};

```

NOTIFY INTERFACE

```

/* NOTIFY_PAUSED */
struct    notify_paused_request {
    data_pause_reason reason;
    u_quad    resvd1;
};
/* No reply */

/* NOTIFY_HALTED */
struct    notify_halted_request {
    data_halt_reason reason;
    string    text_reason<>;
};
/* No reply */

/* NOTIFY_CONNECT */
enum connect_reason {
    NDMPConnected,    /* Connect sucessfully */
    NDMPHostDown,    /* Connection shutdown */
    NDMPConnRefused    /* reach the maximum number of connections */
};
struct notify_connect_request {
    connect_reason reason;
    u_short    ndmp_ver;
    string text_reason<>;
};

```

LOGGING INTERFACE

```

/* LOG_LOG */
struct    log_log_request {
    string    entry<>;
};
/* No reply */

/* LOG_DEBUG */
enum debug_level {
    NDMPuser_info,
    NDMPuser_summary,
    NDMPuser_detail,
    NDMPdiag_info,
    NDMPdiag_summary,
    NDMPdiag_detail,
    NDMPpprog_info,
    NDMPpprog_summary,
    NDMPpprog_detail
};
struct    log_debug_request {
    debug_level level;
    string message<>;
};
/* No reply */

/* LOG_FILE */
struct    log_file_request {
    string    name<>;
    u_short    ssid;
    ndmp_error error;
};
/* No reply */

```

FILE HISTORY INTERFACE

```

/* FH_ADD_UNIX */
typedef      string unix_path<>;
enum unix_file_type {
    NDMPfsdir,
    NDMPfsfifo,
    NDMPfscspec,
    NDMPfsbspec,
    NDMPfsregf,
    NDMPfsslink,
    NDMPfssock
};
struct unix_file_stat {
    unix_file_type    ftype;
    u_long            mtime;
    u_long            atime;
    u_long            ctime;
    u_long            uid;
    u_long            gid;
    u_long            mode;
    u_quad            size;
    u_quad            fh_info;
};
struct unix_fh_entry {
    unix_path    name;
    unix_file_stat file_stat;
};
struct fh_add_unix_request {
    unix_fh_entry unix_fh_entries<>;
};
/* No reply */

/* FH_ADD_UNIX_DIR */
struct unix_dir_ent {
    unix_path    name<>;
    u_long      node;
    u_long      parant;
};
typedef struct unix_dir_ent unix_dir_ent_t;
struct fh_add_unix_dir_request {
    unix_dir_ent_t dirs<>;
};
/* No reply */

struct unix_node {
    unix_file_stat fstat;
    u_long        node;
};
typedef struct unix_node unix_node_t;
struct fh_add_unix_node_request {
    unix_node_t nlist<>;
};
/* No reply */

```