IDRP announce list	each entry:32 list overhead:8 bytes (head and tail pointers)	temporary during parsing and PDU transmittal
gated rt_head per IDRP destination	each entry: xx octets rt_head = xx octets list overhead: ?? (order of 8-16 bytes per family)	Use one per NLRI destination per family (CLNP or IP)
gated rt_entry per IDRP route	each entry: xx octets list overhead: 12 octets (head, tail, active)	Use one per AdjRib entry
IDRP route_in hash table	hash table = hash table entry (16 octets) * hash table size (configured at compile time)	one per Peer
IDRP Outbound Route Array	each entry: 8 bytes number of entries: one per Peer	one per NLRI or idrpRoute structure
IDRP Refresh status structure	24 octets	one per Peer
IDRP Refresh UPDATE PDU structure	each entry: 28 bytes + Withdraw Structure bytes (depends on max PDU size received configured) number of entries: one per UPDATE PDU in refresh sequence	one per BISPDU sent in REFRESH sequence (lots)
idrp_rt_chain_walk structure	entry: 8 bytes list: 8 bytes	temporary - used for search gated routes for route additions or best external route search up to 4 used at a time per NLRI. high water usage per size of routing table will be tested for.
snpa_list	entry: 16 per SNPA list: 12 per list per peer total size: 16 * number of SNPAs + 12	one per peer
gated policy structure (adv_entry)	entry:?? list: ??	one entry per policy statement
IDRP PREF policy structure	entry: ?? list: ???	one entry per PREF statement
IDRP DIST policy structure	entry: ?? list: ??	one entry per DIST policy statement
IDRP AGGR policy statement	entry: ?? list: ??	one entry per DIST policy statement
Temporary policy handling structure	entry: ??	temporary holding structure for policy manipulations

12. Memory organization and sizing information

Memory allocation needs (this table is incomplete, and will be filled in in a forthcoming release of this document):

Type of Structure	Memory Need/structure	Number of Structures used
Peer structure	basic structure: 4K areas of growth: Hash table size configured larger may increase table	one per IDRP Peer, plus one for local node
IDRP Route Structure	basic structure: 88 octets + route_out structure (8/peer) growth: route_out structure grows by 8 bytes per peer	one per AdjRib entry per IDRP received route, one per EXT_INFO entry (IS-IS route), one per Local route
Attribute Record	400 octets for basic structure associated structures of may add growth of 30 per Route ID associated (which may have many NLRI/idrpRoute structures associated)	one per use of attribute structure (can be shared across many peers) note: local routes use idrp_attribute records - one per configured attribute
RouteList in Attribute record	one per Route ID: 30 octets	one per Route_id associated with attribute record
Canonical Rd Path list	entry: 28 octets list: 1 entry per RDI in RD path	one list per attribute record
AS Path	entry: 1 per AS (8 bytes) list: 1 per AS in pathway (overhead 12 bytes)	part of IDRP attribute record: **NOTE:** Only used in interaction with BGP-4 or EGP, ignore for ISO-only implementation
IDRP attribute array	array element: 16 per element array: 1 per IDRP defined attribute	part of IDRP attribute record
User attribute array	array element: 16 per element array: 1 per USER defined attribute supported by IDRP implementation	part of IDRP attribute record
IDRP Error info structure	20 bytes	part of idrpPeer structure -
IDRP send list	each entry:24 bytes plus withdraw Structure (size depends on maximum size of PDU received: maxsize - 30/4) list overhead: 8 bytes (head and tail pointers)	temporary during parsing and PDU transmittal

11. gated log file format

sample Log file:

```
****** Gated Initialization ****

Jun 29 03:31:05 inet_routerid_notify: Router ID: 200.1.1.1
Jun 29 03:31:05
Jun 29 03:31:05 iso_ifachange: Interface 47.0005.80ff.ff00.0000.0400.0000.0000.0000.0000.00
(et1) SystemID 0000.0000.0000
Jun 29 03:31:05 idrp master task now supports idrp pid on the ip raw socket
Jun 29 03:31:05
Jun 29 03:31:05 ***Routes are being installed in kernel
Jun 29 03:31:05
Jun 29 03:31:05 inet_routerid_notify: Router ID: 200.1.1.1
Jun 29 03:31:05
**** IDRP reinit code called to process
**** kernel and static routes from gated
****
Jun 29 03:31:05 idrp_reinit called, but null processing
Jun 29 03:31:05 ph3_status_case2 entered
(** gated interface routes or hsot routes are
*** not added in ph3_status_case2)
***
Jun 29 03:31:05 ph3_status_case2 called peer local_node route Default
** IP default route has IDRP route made for it
Jun 29 03:31:05 ph3_status_case2 entered
Jun 29 03:31:05 ph3_status_case2 entered
Jun 29 03:31:05 ph3_status_case2 entered
Jun 29 03:31:05 ph3_status_case2 entered
Jun 29 03:31:05 ph3_status_case2 entered
*** Here's a 128.2 IP route.
Jun 29 03:31:05 ph3_status_case2 called peer local_node route 80.02
Jun 29 03:31:05 ph3_status_case2 entered
Jun 29 03:31:05 ph3_status_case2 entered
*** IDRP routes flashed to phase 3
****
Jun 29 03:31:05 case1: IDRP peer local_node route 47.0005.80AA.AAAA flashed
Jun 29 03:31:05 case1: IDRP peer local_node route 47.0005.80FF.FFFF.0000.05 flashed
Jun 29 03:31:05 case1: IDRP peer local_node route 49.0128 flashed
Jun 29 03:31:05 ph3_status_case2 entered
Jun 29 03:31:05 ph3_status_case2 entered

(More log file here)
```

```
 */

{

        install_fib(best routes (LOC_RIB))
        DIST_LIST(best routes (LOC_RIB)

}
```

Additional notes on building best external route and best route

Two methods could be used for the first creation of the best external routes or best internal routes: compare all Ribs or compare 1 AdjRib to current best route (null). The second method will be used repeated through out the additional UPDATES received.

```
                                        {
                                        find new best external route
                                        mark global flag that best external route
                                        changed
                                        }
                                }

        /* here comes distribution */

        if best external route changed {
                find all best external route changes and
                send to internal peers
                }

end of phase1 processing
}

/* Phase 2 processing
 *
 */

Scan all AdjRibs associated with internal peers

        for each new route in AdjRib_in
                {
                Is new route better than best internal route
                for this destination?
                        yes - update best internal route mark
                              set best internal route changed

                        no - go on
                }

        for each deleted route in AdjRib in
                {
                if this route is best internal route
                        {
                        select new best internal route
                        mark best internal route as changed
                        }
                }

if (best internal route is better than best external route)
        set best route = best internal route
else
        set best route = best external route
end if

Aggregate routes based on best route;

end of phase 2 processing
}

/* phase 3 processing
```

```
        Look up NLRI in AdjRIB.
        Find out if the NLRI is:
                implicit withdraw for NLRI
                overlapping and more specific with different
                        path attributes

                more specific route
                new route
                overlapping and less specific with different
                        path attributes

        switch (result of NLRI look-up)
                {
                case of withdrawal:
                case of overlapping more specific:
                                {
                                set unfeasible flag to run decision process
                                }


                case of more specific:
                case of new route:
                case of overlapping and less specific:
                                {
                                clear unfeasible flag
                                }
                }
}

/* It is assumed that a BIS maintains for each destination:
 * a best external route
 * a best internal route
 * a best route (LOC_RIB)_
 */

idrp_decision()
{

        /* here is phase 1 */

        Scan all the AdjRib In associated with external peers
                {
                for each new route in AdjRib_in
                        {
                        if new route is external and better than
                        best external route, mark this one as
                        the best external route. Set global flag
                        that best external route changed
                        }

                for each deleted route in AdjRib_in
                        {
                        if route was best external route
```

```
                    while there are more update BISPDU received
                              {
                              get next UPDATE BISPDU:
                              if (process_withdraws (UPDATE) == unfeasible_route
                                        or process_nlri(UPDATE) == unfeasible_route)
                                        {
                                        run idrp_decision process
                                        }
                              else
                                        {
                                        new_route = 1;
                                        }
                              }

                    /* after finish getting all available update PDUs
                     * run decision process.
                     */

                    disable timers;
                    if (new_route == 1)
                              {
                              new_route = 0;
                              run idrp_decision process
                              }
                    }

process_withdraws (BISPDU
{
          for all route IDs in BISPDU
                    {
                    find route_id in Hash table
                    mark withdraw on all route ID's NLRIs
                    place route ID in withdraw array for this peer
                    re-link AdjRib radix structures around the withdrawn
                              NRLI entries

                    if (LOC_RIB list entry was withdrawn)
                              {
                              flag unfeasible route so decision process will run
                              }
                    if (best external route was withdrawn)
                              {
                              flag unfeasible route so decision process will run
                              }

                    if (aggregated affected by withdrawn)
                              {
                              flag unfeasible route so decision process will run
                              }
                    }
}

process_nlri(UPDATE)
{
```

10. Alternative designs

In researching our design choices, we investigated different data structures, and UPDATE PDU processing logic that would be effective under a multi-threaded environment. Section 9.1 describes some of the other alternatives for Data Structure. Section 9.2, provides logic for the UPDATE PDU processing which minimizes the times the decision process needs to be run. AdjRib updates are considered to be done in an second process. The second process locks the AdjRibs it is updating.

10.1. Alternative data structures for AdjRib and Loc_RIB

One approach to the AdjRibs, and LOC_RIB is to build a linked list per peer of NLRI destinations. The search of the routes is done by sequential search. The original Merit prototype code used this ineffective sequential search. However, it was not considered for our next stage prototype because of the costly nature of the look ups.

Each AdjRib structure can be built as a radix trie with additional information pointers. The best external routes would be generated as another radix trie with pointers into different AdjRibs. The LOC_RIB would also be another radix trie with pointers into different AdjRibs.

Benefits of each AdjRib in radix trie:

1) quick updates to AdjRib which can be done by independent processes

2) Overlaps easily recognized per Peer, so best fit returned easily.

3) Removal of routes per peer involves simply releasing this radix trie structure

4) Refresh cycle can build whole AdjRib prior to swapping it in as the current routing.

Concerns:

1) Links between Adjacent Ribs must be maintained

2) How do IS-IS routes interact with this radix trie by destination structure?

10.2. Alternative UPDATE logic

One approach to damping out the oscillations the minimum route selection timer tries to fight is to run the decision process at least as frequently as the Maximum Decision Internal, and at most as frequently as the Minimum Decision interval. By only doing the decision process at intervals, you damp out the route oscillations. This global timer is not in the specification, but provides the same functionality with considerably less cost than the IDRP specification.

If a UPDATE process used this approach, the following logic could be used in the main loop processing UPDATE BISPDUs.

Logic:

```
main()
{
        while (1) {
```

SIGHUP	calls task_reconfigure (unless no-reconfigure bit is set for the task) to call protocol to reconfigure	master task - idrp_cleanup peer tasks idrp_peer_cleanup handles gated re-reading configuration file. Policy lists are freed and the following initialization sequence is followed: idrp_cleanup idrp_var_init parsing the gated file idrp_init routine idrp_flash - first update idrp_newpolicy (won't be implemented until policy filters implemented in 2nd phase of project).
SIGUSR1	toggle tracing flag	none
SIGUSR2	call if_check to see if any new interfaces have been added or changed.	none
SIGCHLD	calls task_child	none
all other UNIX interrupts	ignored by gated	none

9. Gated interrupt signals

Gated is driven in two ways - signals and polling. Signals are used for reconfiguring gated routing policy or interface configurations, or to change what logging is used. The PDUs received by gated are received by a select in task_main that blocks until a socket is ready for reading or writing.

The gated code catches UNIX system kills, alarms (for timers), USR1 and USR2, INT, HUP and CHLD. Incoming PDUs do not interrupt gated. The gated code (in task.c) simply does a “select” on the socket and waits for one or more them to be ready to read or write. The following task signals are handled by gated. Below is a chart of the gated signals and the IDRP routines called to handle the gated function.

Task Signal	task.c gated routine	IDRP routine called
SIGTERM	task_terminate - called to let each protocol shut down gracefully	idrp_terminate (gracefully terminate master task for PDU reception) idrp_peer_terminate (terminates task per IDRP Peer)
SIGALARM	timer_dispatch called to find out what timers are pending and start up the respective tasks	master task timers - none peer task timers: Keepalive timer, CloseWait timer, Holdtime timer, Retransmit timer, IDRP start up timer, open sent timer, min_adv timer, IDRP echo timer (associated with debug feature to echo BISPDU packets)

IDRP_MASTER_TIMER _MIN_ADVRD	Timer for the Minimum route advertisement timers for routes local to this RD	timer_create timer_reset timer_set

growth factor: Withdraw structure directly impacts send list

8. Gated timer functions usage by IDRP

IDRP creates the following timers for each peer task: Keepalive, closewait, hold, retransmit, start, Open sent, Minimum Route Advertisement, and optional debug echo timer. The Open sent timer allows for multiple opens to be sent in rapid sequence prior to holding down the connection until the next open sequence. The optional debugging echo function allows echoing at the BISPDU level.

IDRP Timer	Function	Gated routines used to handle it
IDRPTIMER_KEEPALIVE	Keepalive timer	task_create (idrp_init) timer_set (start_keepalive_timer) timer_reset (close_peer, begin_close)
IDRPTIMER_CLOSEWAIT	CloseWait timer	task_create (idrp_init) timer_set (begin_close) timer_reset (close_peer)
IDRPTIMER_HOLDTIME	Hold Timer	task_create (idrp_init) timer_set (idrp_recv_pdu, idrp_start_event) timer_reset (close_peer, begin_close)
IDRPTIMER_REXMIT	Retransmit Timer	timer_create (idrp_init) timer_set (start_rexmit_timer) timer_reset (kill_rexmit_timer)
IDRPTIMER_START	IDRP start timer	timer_create (idrp_init) (note created with timer value to expire) timer_reset (idrp_event_starttimer)
IDRPTIMER_OPENSENT	IDRP OPEN sent code	timer_create (idrp_init) timer_set timer_reset (close_peer)
IDRPTIMER_MIN_ADV	IDRP Minimum Advertisement Timer	timer_create (idrp_init) timer_set (phase3_ext_send) timer_reset (begin_close)
IDRPTIMER_ECHO	IDRP Echo timer for Echo debug function	timer_create (idrp_init)

Timer for only the local Node

any idrpRoute structure (p_p_best_ext->p_best_ext) we can discover the best external route to the NLRI referenced by that structure.

The idrp_best_ext structures themselves are kept in an unsorted linked list of idrp_best_ext structures. In effect, these form what we call a “best external routes RIB,” or BER_RIB. Having the BER_RIB accessible faciliatates sending best external routes to newly activated internal peers.

Inbound Route ID table

The IDRP route ID look up uses a simple hash of the route ID look up divided by a constant.

Outbound Route ID Table

A Outbound Route list table exists for every NLRI (idrpRoute structure). Since a different Route_id could be used for each exterior peer, a route id table is indexed by peer. For each BISPDU sent with a route_id to a peer, the list of NLRIs in that BISPDU are linked on a circular link lists tagged by the route ID. Each idrpRoute structure contains a table where these lists are linked by peer.

Auxiliary Lists used for IDRP

The attribute link listed is a linked list with forward and backward pointers. The route ID lists associated with the attribute record are simple linked lists.

For overlapping routes, the basic implementation of the IDRP protocol will import all routes. As policy is implemented in the second phase of development, the basic gated routines will be added to give best match as well as exact match in the radix tree look-up code.

7.6. MD4 algorithm

The MD4 algorithm is described in RFC 1320. The code in our implementation is taken from the reference implementation included with that RFC. The MD4 algorithm has been encoded but no testing has been done with it at this time. By Delivery 1, some testing will be done on this encryption plus a tool developed that will allow PDUs to be decoded on the wire. The tool will be tcpdump with added support.

7.5. Route look up algorithms

AdjRib and LOC_RIB

The Adjacent RIBs are "virtual" which means the Adjacent RIBs are not unique structures. Instead, the structures are a thread through the normal gated routing table. You can request an Adj-Rib by specifying the peer and the IDRP protocol as a look up parameters into the gated table.

The LOC_RIB for gated contains not only the IDRP routes, but external routes as well. Gated calls the LOC_RIB the list of active routes. These active routes are installed in the operating system (or kernel) of the router.

Gated uses the radix trie for the Route look-up on destination. Its routing structures are primarily based on destination address or NLRI.

BER_RIB

In order to support the tracking and advertisement of best routes learned from external peers which, however, are not necessarily the best routes (that is, some route learned from an internal peer may have a better preference), we introduce several data structures. The structures used for tracking best external routes information are pictured below:

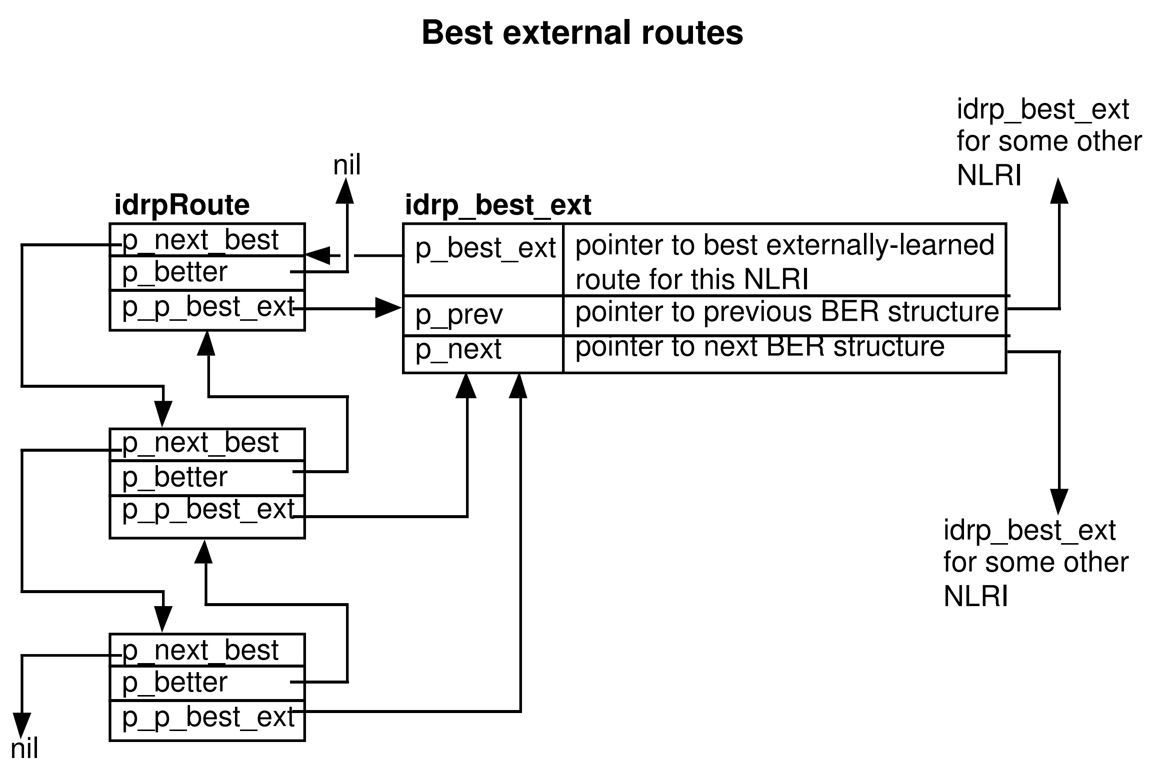


Figure 32 — Best external routes data structures

We use the p_better and p_next best pointers in the idrpRoute structure to maintain a linked list of NLRI to the same destination, sorted by IDRP preference. By doing a double indirection from

```
route_id      INTEGER(1...4294967295);  # 4 byte integer
loc_pref      INTEGER(1..255();                # 1 byte integer
}
```

ASN.1 needed:

```
countLocRibs:: = INTEGER(1...255);
countRIBatts ::= INTEGER(1..255);

AdjRibIdset ::= SET OF ribid;
LocRIBIdset ::= SET OF ribid;

AdjRibs ::= SEQUENCE OF Rib;
LocRibs ::= SEQUENCE OF Rib;

Rib :: = SEQUENCE OF
        {
        Ribid   ribid;
        ribroutes       RibRecords;
        }

ribid ::= INTEGER (0..255); - where 0 is the default rib
RibRecords :: = SET of RibRecord;

RibRecord :: = SEQUENCE OF
                {nlri NLRI,
                 pathattmask    PathRibAttributeMask,
                 pathatt PathRibAttributes;
                }

PathRibAttributes ::= SEQUENCE OF PathRibAttribute;
PathRibAttribute ::= SEQUENCE OF {
                nonDistAtt      NonDistAtt;
                nonDistval      NonDistvalue;
                }


NonDistAtt ::= ENUMERATED {
        ROUTE_SEPARATOR (1),
        EXT_INFO (2),
        RD_PATH (3),
        NEXT_HOP(4),
        DIST_LIST_INCL(5),
        DIST_LIST_EXCL(6),
        MULTI_EXIT_DISC(7),
        HIERARCHIALRECORDING(12),
        RD_HOP_COUNT(13)};

NoDistAttValue ::= CHOICE OF {
        routeSepInfo[0]         RouteSepInfo;
        OnPathVal[1]  BOOLEAN;
        rdPath[2]       Set of Rdi;
        IntPathValue[3]         INTEGER(1..255);                # 1 byte integer
        nextHop[4]      NextHop;
        }

RouteSepInfo :: = SET of {
```

BEHAVIOR DEFINED AS The count of LocRibs in use by this BIS. A Loc Rib is distinquished by the Distinquishing Rib Atttributes.";;
REGISTERED AS {IDRP.atoi countLocRib(37));

locRibidSet ATTRIBUTE;
WITH ATTRIBUTE SYNTAX IDRP.LocRibIdSet;
BEHAVIOR LocRibIdSet-B
BEHAVIOR DEFINED AS The count of mapping of AdjRib Distinquishing attributes to the local node's AdjRib structure. ";;

locRibs ATTRIBUTE
With ATTRIBUTE SYNTAX IDRP.LocRibs;
BEHAVIOR AdjRib-b
BEHAVIOR DEFIED AS the set of LocRibs associated with this BIS.
Each LocRib is ordered by NLRI.";;

DERIEVED from “Rec. X.72 | ISO/IED 1065-2: 1992”: top;
CHARACTERIZED BY idrpAdjRibPkg PACKAGE
BEHAVIOR idrpAdjRib-B
    BEHAVIOR DEFINED AS The information AdjRibs associated with the BIS. One managed object exists for each Adjacent BIS. Index by bisNet.”.

Attributes:
    bisNet GET,
    countRIBatt GET,
    AdjRibIdset GET,
    AdjRibs GET;

countRibAtt ATTRIBUTE;
WITH ATTRIBUTE SYNTAX IDRP.countRibAtt;
BEHAVIOR countRibAtt-B
    BEHAVIOR DEFINED AS The count of AdjRib associated with this BIS. An AdjRib is distinquished by the Distinquishing Rib Atttributes.”;;
REGISTERED AS {IDRP.atoi countRibAtt(37));

AdjRibIdSet ATTRIBUTE;
WITH ATTRIBUTE SYNTAX IDRP.AdjRibIdSet;
BEHAVIOR AdjRibIdSet-B
    BEHAVIOR DEFINED AS The set of local RibSetId that represent AdjRibs that this adjacent BIS has sent to this neighbor.”;;
REGISTERED AS {IDRP.ato AdjRibIdSet(38));

AdjRibs ATTRIBUTE
With ATTRIBUTE SYNTAX adjRibs;
BEHAVIOR AdjRib-b
    BEHAVIOR DEFIED AS the set of AdjRib associated with this BIS. It may be ordered by AdjRib internal id or nlri. “;;

REGISTERED AS {IDRP.ato AdjRibs(39));

idrpLocRib MANAGED OBJECT CLASS
DERIEVED from “Rec. X.72 | ISO/IED 1065-2: 1992”: top;
CHARACTERIZED BY idrpLocRibPkg PACKAGE
BEHAVIOR idrpLocRib-B
    BEHAVIOR DEFINED AS The information the LocRib associated with the local BIS. One Loc_RIB exists for each Rib Attribute plus the Default associated with a BIS”.

Attributes:
    countLocRibs GET,
    LocRibIdset GET,
    LocRibs GET;

countLocRibs ATTRIBUTE;
WITH ATTRIBUTE SYNTAX IDRP.countLocRib;
BEHAVIOR countRibAtt-B

```
                    preflgth NSAPPrefixLength;
                    prefix NSAPPrefix,
                    secigth SecurityLength,
                    secval SecurityLevel}

9) SecurityLength :: = INTEGER(0..255)

10) SecurityLevel ::= OCTETSTROING (SIZE(1..255))

11) QOS ::= CHOICE { global[0] EXPLICIT GLOBAL,
                     ssQOS[1] EXPLICIT QOSTV,
                     dsQOS[2] EXPLICIT QOSTV}

12) GLOBAL ::= ENUMERATED {
                    delay(0),
                    expense(1),
                    capacity(3),
                    error(4)};


12) QOSLength ::= INTEGER(1..255)

13) QOSTV ::= SEQUENCE {
                    preflgth NSAPPrefixLength;
                    prefix NSAPPrefix,
                    QOSlgth QOSlength,
                    QOSval QOSValue };

14) QOSValue ::= OCTET STRING(SIZE(1..255));

15) Ribattribute ::= ENUMERATED {
                    tRANSITDELAY(9),
                    rESIDUALERROR(10),
                    eXPENSE(11),
                    locallyDefinedQOS (12),
                    security(14)
                    capcity(15),
                    priority(16)}
```

7.4.4.3. Replacement GDMO for ATN project's LOC_RIB and AdjRIB

The GDMO for ATN asks for the AdjRibs, LocRIBs and the locFIB. The LocalFIB is a matter outside the gated user program. global matter. However, for reasons of debugging

and use of IDRP with upcoming source routes the ability to pull copies of these AdjRibs and LocRibs may be useful.

The adjRib, LocRibs and LocFIBs should not be required of every router. To get simple IDRP routers, the local dump of this information may be more cost effective.

GDMO for AdjRib and LocRib:

idrpAdjRib MANAGED OBJECT CLASS

7.4.4. GDMO for this MIB

7.4.4.1. GDMO in the IDRP specification

For the most part, the idrpConfig and adjacentBIS managed objects were printed out from the Peer structures.

One note about the Intra-IS variable. Each internal configured route will be configured with the next_hop address within the domain. Each route imported from another protocol will also have a next hop gateway. Therefore the list of IntraIS exists within gated as either the configured set of local peers for each local route, or the learned IS via IS-IS or other Intra-Domain protocols.

This variable has been left out of the MIB DUMP for now.

7.4.4.2. GDMO imported and clean-up from IDRP specification

```
1) NLRI ::= NSAPPrefix
2) NSAPPrefix ::= BIT STRING(1...160)
3) SystemIdGroup ::= Sequence {
                nETs Set of NETPrefix
                nSAPS set of ESPrefix
                }

4) ESPrefix ::= NSAPPrefix
```

Definitions for Attribute Ribs and AdjRibs and FIBs

```
1) RibAttSet :: = SEQUENCE {
                        confed RibSetId
                        count RibSetcount,
                        attribs SET OF Ribattributes}

2) Ribatt ::= SEQUENCE {
                        attrib          SET OF RibAttribute,
                        value           SET OF RibValue OPTIONAL,
                        }

3) RibSetId :: = INTEGER(1...255)

4) RibSetCount ::= INTEGER(0..255)

5) Ribattributes ::= SEQUENCE {
                priority [0] EXPLICIT Priority OPTIONAL,
                security [1] EXPLICIT SEC OPTIONAL,
                qosmaint [2] EXPLICIT QOS OPTIONAL }

6) Priority ::= INTEGER(0..14)

7) SEC ::= CHOICE [ ssDEC[0] EXPLICIT Ribattsec,
                 dsSEC[1] EXPLICIT Ribattsec }

8) RibAttSec :: = SEQUENCE {
```

```
RibID Path id Destination Next Hop NET
0 4 49.0129  (null)
0 5 49.0133  (null)
0 6 Default  (null)
0 6 80.03    (null)
```

```
IDRP Attributes:
 #0 ATTR RefCnt 1, RIB_id 0, Mask 100d
 IDRP_ATTR_ROUTE_SEPARATOR RouteID 0 Local Pref 0
 IDRP_ATTR_RD_PATH
        49.0130
 IDRP_ATTR_NEXT_HOP 47.0005.80ff.ff00.0000.0400.0000.0000.232a.0161.00
 IDRP_ATTR_HOP_COUNT 0 0
 #1 ATTR RefCnt 2, RIB_id 0, Mask 100d
 IDRP_ATTR_ROUTE_SEPARATOR RouteID 0 Local Pref 0
 IDRP_ATTR_RD_PATH
        49.0130
 IDRP_ATTR_NEXT_HOP 47.0005.80ff.ff00.0000.0400.0000.0000.2301.0172.00
 IDRP_ATTR_HOP_COUNT 0 0
 #2 ATTR RefCnt 1, RIB_id 0, Mask 100d
 IDRP_ATTR_ROUTE_SEPARATOR RouteID 0 Local Pref 0
 IDRP_ATTR_RD_PATH
        49.0130
 IDRP_ATTR_NEXT_HOP 35.42.1.125
 IDRP_ATTR_HOP_COUNT 0 0
 #3 ATTR RefCnt 1, RIB_id 0, Mask 100d
 IDRP_ATTR_ROUTE_SEPARATOR RouteID 0 Local Pref 0
 IDRP_ATTR_RD_PATH
        49.0130
 IDRP_ATTR_NEXT_HOP 35.42.1.33
 IDRP_ATTR_HOP_COUNT 0 0
 #4 ATTR RefCnt 1, RIB_id 0, Mask 100d
 IDRP_ATTR_ROUTE_SEPARATOR RouteID 0 Local Pref 0
 IDRP_ATTR_RD_PATH
        49.0129
        49.0130
 IDRP_ATTR_NEXT_HOP (null)
 IDRP_ATTR_HOP_COUNT 1
 #5 ATTR RefCnt 1, RIB_id 0, Mask 100d
 IDRP_ATTR_ROUTE_SEPARATOR RouteID 1 Local Pref 0
 IDRP_ATTR_RD_PATH
        49.0129
        49.0130
 IDRP_ATTR_NEXT_HOP (null)
 IDRP_ATTR_HOP_COUNT 1
 #6 ATTR RefCnt 2, RIB_id 0, Mask 100d
 IDRP_ATTR_ROUTE_SEPARATOR RouteID 2 Local Pref 0
 IDRP_ATTR_RD_PATH
        49.0129
        49.0130
 IDRP_ATTR_NEXT_HOP (null)
 IDRP_ATTR_HOP_COUNT 1
```

RIB contents are in the routing table:

ISO routes for idrp are :

```
      IDRP Local Rib:
 RibID Path id Destination Next Hop NET
 0 6 80.03    (null)
 0 1 47.0005.80AA.AAAA     47.0005.80ff.ff00.0000.0400.0000.0000.2301.0172.00
 0 1 47.0005.80FF.FFFF.0000.05    47.0005.80ff.ff00.0000.0400.0000.0000.2301.0172.00
 0 0 49.0128  47.0005.80ff.ff00.0000.0400.0000.0000.232a.0161.00
 0 4 49.0129  (null)
 0 5 49.0133  (null)

      IDRP ADJ Rib:
 Net: 47.0005.80ff.ff00.0000.0400.0000.0000.232a.0144.00 Rib Id: 0 (default)
```

```
IDRP Peer MIB structures :

 bisNegotiatedVersion 1
 bisNet 47.0005.80ff.ff00.0000.0400.0000.0000.232a.0144.00
 bis{eerSNPAs: 7d.e1.ce.3a.0
 bisRDC: No RDC support
 bisRDI 49.0129
 KeepAliveSinceLastUpdate 40
 lastAckRecv (send_unack) 7
 lastAckSent (rcv_seq_expected - 1) 5
 lastSeqNoRecv (rcv_seq_expected - 1) 5
 lastSeqNoSent (send_next -1) 6
 listForOpen TRUE
 MaxPDUPeer 200, OutstandingPDUs (recv_cerdit) 5
 State IDRP_ESTABLISHED
 totalBISPDUsin 0, totalBISPDUsout 0, updatesIn 0, UpdatesOut 0
 /* SND.NXT = 7, SEND.UWE = 12, SND.UNA = 7
 * RCV.EXP = 6, RCV.CRED = 5, RCV.CREDAV = 5
 * HOLDTIME = 30, KEEPALIVETIME = 10
 */ MAXSENDSIZE = 200, MAXRCVSIZE = 200
```

```
MaxRibIntegrityCheck 0, MaxRibIntegrityCheckTimer 0
MinRouteAdvTimer 00:00:30, MultiExit FALSE, Priority 0
rdcConfig: No rdc support
rdLRE: Not supported
RetransmissionTime 0
ribAttsSet: Only default routes supported
RouteServer OFF, Version 0
```

(These structures will be designed later. This section will contain information about IDRP specific structures and the gated structures used by IDRP.)

7.4. Network management information base

7.4.1. Overview

Network management information is dumped as part of a call to the SIGINT call. This call cause IDRP to dump certain things depending on the parameter. Currently the MIB only dumps IDRP GDMO format. In Delivery 2, one of parameters will specify the parameters listed in the IDRP GDMO, another will specify the IDRP for IP MIB.

The input to the MIB information is done via the configuration file. Please see the configuration format below for the MIB definition. Network Management variables are set via the configuration line and changed via the SIGHUP signal to gated. The tracing or dump parameters for the MIB may also be changed via the SIGUSR1.

In the future, a proxy-SNMP/CMIP agent could be added to gated that would talk to a real SNMP/CMIP agent. Or a CMIP agent could be added to the code.

7.4.2. MIB input structures

Pre-Delivery 1 and Delivery 1:

idrpConfig variables are either pre-configured or added via the local_node configuration parameters. The Adjacent BIS parameters are added mostly in the configuration parameters, and some in the IDRP configuration.

Delivery 2:

idrpConfig structures are included in the local_node configuration format. Adjacent BIS parameters are available via the configuration file. All policy is available via the configuration file.

7.4.3. MIB output structures

Below is a sample output of a the MIB information.

```
IDRP Master Task MIB structures :
 AuthenticationTypeCode 0, Capacity 0, CloseWaitDelayTime 0
 ExternalBisNeighbor:
 NET 47.0005.80ff.ff00.0000.0400.0000.0000.232a.0144.00 (35.42.1.68)
 HoldTime 0
 InternalBis:
 InternalSystems:
 49.0128
 47.0005.80FF.FFFF.0000.05
 47.0005.80AA.AAAA
 Default
 80.02
 IntaIS:
 Not Implemented
 KeepAliveTime 0, localRDI 49.0130
 Local SNPA: 55.d7.b0.12.0
 LocExpense 0, MaxCPUOverloadTime 0
 MaxPDULocal (pdu_maxrecvsize) 200
```

```
#per Steve Hotz
#
preference_list {
       PREF <route_pattern> < local_cond> = "value";
       }

route_pattern ::= <nlri> <info_src> <path> <dist_att> <attrib_cond>

nlri ::= '{' <dest_list> '}' | <id_string>#id-string is name for destination
<dest_list> ::= <dest> ',' <dest_list> | <dest>
<dest> ::= "any" |
        "not" <nsap> ':" '*' |
        "not <nsap>" |
        "not" <IP_address> [IP_mask];

info_src :: = "idrp" | "ext_info" | "any" | "bgp" | "egp" | "rip" | "ospf" | "isis"

#
# Distribution lists
#
d_stmt_list ::= <d_stmt> <d_stmt_list>

d_stmt ::= "DIST" <route_pattern1> <local_cond> '='
              [<bis>] <select_action> '{' <mod_list> '}' "DONE | "CONT"

bis ::= <adj_bis> | "local" | intra_domain";
```

```
		authType <integer>
		authCode <hex string - 16 digits>
	# Which protocol nlri will this support
		proto {ip | clnp}
	# Which protocol socket does it run over
		proto_sock {UDP | IP | CLNP | IDRP }

		neighbor 0x490128
		 0x47000580ffff00000004000000000232a014400 35.42.1.68
		 intf 35.42.1.85;
		neighbor 0x49003501
		 0x47000580ffff00000004000000000232a01b400 35.42.1.180
		 intf 35.42.1.85;
	};
	external
	{
		# neighbor <neigh-RDI> <neigh-NET> <neighb-IP> intf <our-IP> snpa
		# <octetstring> authcode <integer>
	}
	# key words to same thing: idrp_local osi_local idrp_internal_systems
	#

	idrplocal{
		# all of these are third-party announcements
		# <prefix> <NET-of-next-hop> {<snpa1>} {<snap2>} {<snpa3>};
		#

		0x490128		0x47000580ffff00000004000000000232a014400;
		0x47000580fffff000004 0x47000580ffff00000004000000000232a014400;
	};

};

isis ISO
{
	level 2;
	traceoptions all lspcontent;
	# systemid is not necessary I believe, IS-IS will discover it
	systemid <0000232a0155>;
	# area is not necessary I believe, IS-IS will discover it
	area <47.0005.80ff.ff00.0000.0400.0004>;
	circuit <et0>
		metric 10 priority 20
		metric level 2 20 priority level 2 20;
	prefix internal metric 45 <47.0005.80ff.ff00.0000.0400.0004>;
	prefix internal metric 45 <47.0005.80ff.ff00.0000.0400.0032>;
	prefix external metric 10 <47.0005.8000.5500.0000>;
};

static {
	default gw 35.42.1.33;
};

import proto idrp <rdi> {
	preference <default_preference>
	{
	<preference_list>
	}

aggregate ??? <aggregation_list>

export proto idrp <rdi> <neighbor> {
	<distribution_list>
	}
```

```
tracefile "/tmp/gatedlog.idrp" replace size 100k files 3;
#traceoptions update internal external update route idrp kernel isis;
traceoptions internal external route update idrp;
as 281;
# our RDI
rdi 0x490128;
# our NET (what if we have two NETs?)
net 0x47000580ffff000000040000000000232a015500;
bgp no;
redirect no;
egp no;
rip no;
# how do we config IDRP to run over CLNP vs. UDP?
idrp on {
       traceoptions idrp;
# more trace options will be defined here in Delivery 2
#
       localnode {
# Delivery 1 - will support only
# the non-commented variables
#
#       RIB_ATT_SUPPORTED = "Default" - from STeve's paper
#       BISNET {hex or dot format}
#       intra-is    net;
#       rdc         0x47000580ffff00000008;
#       as_rdi      0x47000580010203040507;
#
        snpa  0x804030201043;
        protocols_supported {ip | clnp | sip | pip};
        proto-sock { ip | iso };
        route_server_allowed {yes | no};
        multi_exit_disc_used {yes | no};
        authType           integer;
        AuthType           integer AuthCode 0x12345678
#
        hold_time   integer;
        outstanding_pdu    integer;
        max_pdu_size 2000;
        listenopen  {yes | no}
#
        maxRibCheck integer for seconds;
        MinAdv      integer for seconds;
        MinAdvRD    integer for seconds;
#
        holdtime    integer for seconds;
        keepalive   integer for seconds;
        rexmit      integer for seconds;
        closewait   integer for seconds;
#
# QOS stuff goes here for local node
#
        }
       group
       internal
       {
       # must have either neigh-ip or neigh-NET or both plus RDI
       # neighbor <neigh-NET> <neigh-IP> rdi <neigh-RDI>
       #
       # interface - should be specified for now
       #       but later can be discovered
       # interface by IP address or SNPA from ES-IS cache
              intf <our-IP>}
              snpa <octetstring>
       # Authentication stuff
```

7.3.2. Pre-delivery 1 policy

Current policy allows only the setting of next hop on the static gated or IDRP routes. Code exists to add other options, but needs further debugging. ISO support in the AIX or BSD/386 systems for IDRP over CLNP PDUs is still being worked on. Therefore setting the ISO protocol is likely to encounter problems.

7.3.3. Delivery 1 policy

7.3.3.1. Policy structure on routes

Temporary parsing structure within the IDRP parsing code is the idrpRoute_option structure which has contains optional policy for a locally configured route:

idrpRoute_options:

1) rib_id
2) SNPA list
3) multi_exit value for route
4) list of RDIs for DIST_LIST_INCL
5) list of RDIs for DIST_LIST_EXCL

In addition, each locally configured (INTERNAL_SYSTEMS) IDRP route must always have the route and the NET of the NEXT_HOP.

These locally configured routes allow the first version to set:

- Next HOP SNPAs for a route
- MULTI_EXIT_DISC
- DIST_LIST_INCL
- DIST_LIST_EXCL

Additionally the following information is configured via the gated configuration syntax. Please note the changes to the gated syntax document to allow for gated syntax.

7.3.4. Delivery 2 policy

The second Delivery policy will implement version 3.0 and up of the gated syntax description language. Please reference the gated syntax document for further details.

7.3.5. Notes on current policy

7.3.5.1. Indirectly listed:

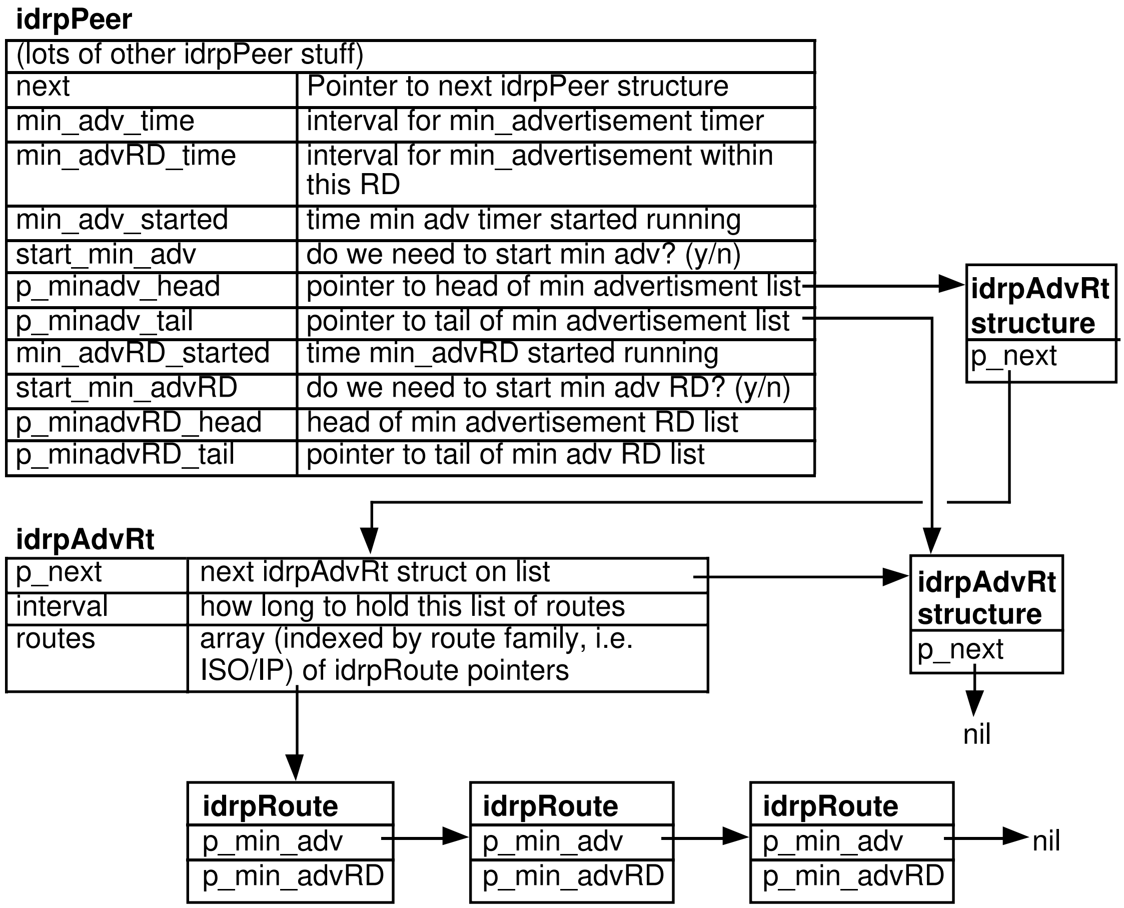
IntraIs - these are neighbors which IDRP can deliver NPDUs for routes for routes local to this domain. These are gateways statically configured on static routes or listed as a protocol gateway.

7.3.5.2. Configuration file format

Sample interim configuration file

```
#
# jgs -- isis config
#
#yydebug yes;
```

Advertisement timer structures



In this example, we illustrate the links for the min adv timer. The links for the min adv RD timer are not shown but might exist (linking a potentially different set of structures). Note that there are in fact a set of idrpRoute lists, one per route family (so in the present case there would be an IP and an ISO list).

Figure 31 — Advertisement Timer Structure

7.3. Policy information base structures

7.3.1. Overview

The basic IDRP protocol router support (pre-delivery 1) has minimal policy and a lot of hard-coded configuration information. The major function to be added to gated is the actual configuration of local routes to test:

- SNPA
- NEXT_HOP
- DIST_LIST_INCL
- DIST_LIST_EXCL

In Delivery 2, the IDRP code will follow the general formats of the “IDRP gated syntax document version 3.0).

7.2.3. IDRP peer types

There are four IDRP Peer types: external, internal, test, and local. The external BIS peer and internal BIS peer are defined in the IDRP protocol specification. In additional, each local node configuration requires a "local" node (or peer) configuration. Only one of these structures exists per gated daemon.

The unique type of peer is a "test" peer. This peer will be a peer which receives all routing information, but is not expected to send routing information. The purpose of the "test" peer is to provide a peer that serves as a recorder of routing information sent in the BISPDUs. This concept is fully worked out in the BGP code, and will be borrowed after delivery 2.

7.2.4. idrp_peer list

All idrpPeer structures are linked on a single list of idrpPeers. However, this listed is

7.2.5. idrpAdvRt structure

The idrpAdvRt structure has the structure pictured below. This structure allows the minimum advertisement code to link routes together at the end of each phase 3 processing that are sent out. These routes are linked to the linked list of idrpRoutes linked through the idrpRoute structure link for the minimum route timer that is running (either p_min_adv for remote RDs or p_min_advRD for local RD routes).

The routes are linked to the list for the family, and therefore the routes structure is an array of linked list structures.

RibRefresh Processing

9) Minimum route Advertisement Timer

10) AdjRib Checksum information

Of these structures, the structures inbound hash may take up the most space depending on the configured size. To limit space on a connection, the hash table can be shrunk. The rest of the configuration and state information is required by:

• IDRP specification
• gated interaction
• tracing information (from an operations viewpoint, this is not a place to cut space!)
• multi-protocol support.

Depending on the feedback of tests for IDRP code Delivery 2, the issues of shrinking the idrpPeer structure may be revisited.

Most of the idrpPeer structures are either gated structures, or timers, flags or defined in the IDRP GDMO. Of the gated structures, the gw or gateway structure may be confusing. This structure is described in the Policy portion of the data structures.

Four idrpPeer structures deserve special mention:

1) Peer status flags (see table below)
2) type of Peer
3) idrp_peer_lists
4) idrpAdvRt structure used for Minimum Route Advertisement lists. (see figure 31)

7.2.2. idrpPeer status flags

The Peer status flags allow the IDRP Code to handle the gated initialization, reconfiguration and deletion of idrpPeers. The initialization, reconfiguration and terminate code make use these flags. The table below summarizes the status flags and their use in the IDRP code.

Status Flags relating to IDRP Peer

Status of Peer	Definition
IDRPF_UNCONFIGURED	zero value, means configuration gave incomplete values since something should be set in the Peer structure
IDRPF_DELETE	Delete this peer since it has been deleted from the configuration. The reparsing routines set this flag on each Peer structure prior to parsing the new configuration file. If this flag is not cleared by a configuration line for this peer it is deleted.
IDRP_TRY_CONNECT	Trying to connect to this IDRP peer but the connection is down.
IDRP_CONNECT	IDRP peer is connected via IDRP.
IDRP_WRITEFAILED	IDRP code tried to write to task socket and failed. gated or the socket support is failing for news for the this peer’s communication.
IDRPF_IDLED	This IDRP Peer is permanently idled by the configuration line.

Output Buffer for IDRP PDUs

idrpBuffer

next	next output buffer
seq	sequence number of this buffer
pdu_proto	pointer to variable-length protocol-specific header
proto_len	length of protocol-specific header
pdu	pointer to BISPDU
length	length of BISPDU
sequenced	is this BISPDU sequenced? (y/n)
ref	refcount

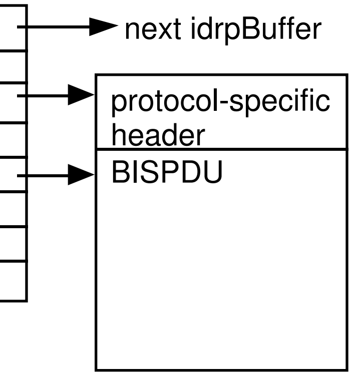


Figure 30 — Output Buffer for IDRP

7.2. IDRP peer structure

7.2.1. Overview

The idrpPeer structure is used for two different types of Peers, the local peer and the adjacent BIS peer. While some space can be saved by making these two structures, in Delivery 1 and Delivery 2 the idrpPeer structure will used for both.

This decision will be re-examined prior to the delivery of the IDRP routes with aggregations.

The idrpPeer structure has the following types of information:

1) links to other peer structures

2) Configuration IDRP specific parameters as found in the idrpConfig or adjacentBIS Managed Object structure.

3) State and Timer information as found in idrpConfig or adjacent BIS Managed Objects

3) additional configuration for multi-protocol support such as IP address and gateway information and types of protocols supported

4) gated gateway information. A gateway structure for the protocols supported must exist to add routes to gated and execute the policy routines.

5) gated task and tracing information pointer to gated task structure tracing information status of peer

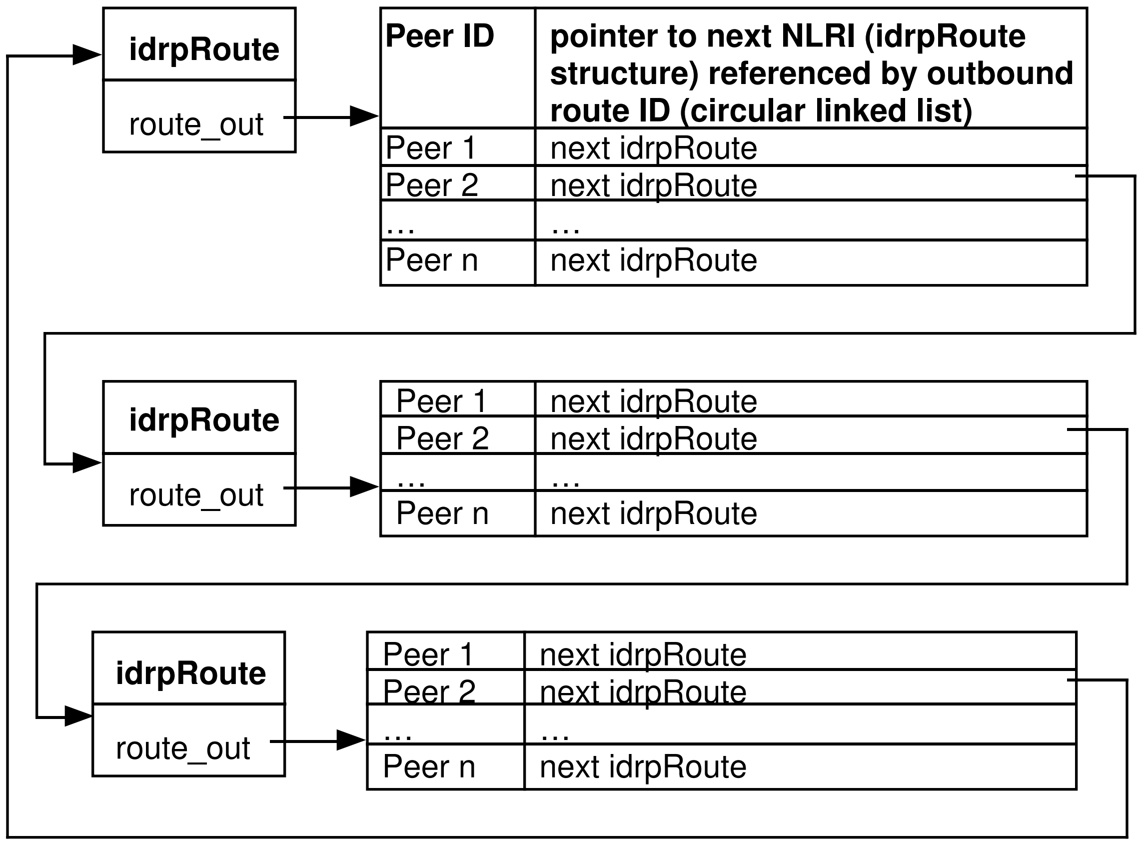
6) Route_id information (current inbound and outbound)

7) hash table for inbound route_ids

8) PDU process information

Error PDU processing

Outbound route ID list



Here we picture the manner in which we would link NLRI associated with the same outbound route ID for an update sent to Peer 2. In the case of Peer 2, there are three NLRI (three idrpRoutes) which have been associated with a single Route ID. Route IDs associated with other peers might result in different chains.

Figure 29 — Outbound route ID list

Send List

idrp_send_list

p_next	next structure on send list
p_attr	attribute record associated with entry
rib_id	RIB ID for routes in this list
pdu_bytes	number of bytes used for NLRI and withdraw ID
pdu_bytes_attr	number of bytes used for attributes
loc_pref	local preference
ann_nlr	array (by route family, i.e. ISO/IP) of pointers to idrpRoute structures, holding NLRI to be announced
withdraw	withdrawal structure, contains count of routes to be withdrawn and IDs of routes to be withdrawn

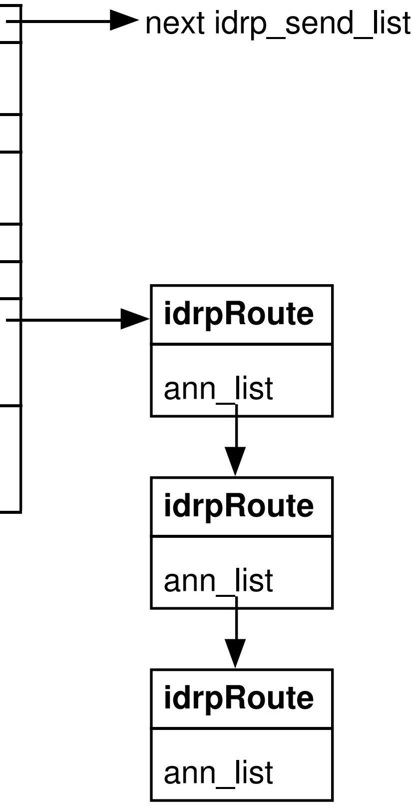


Figure 28 — Send list

Withdraw route linked list

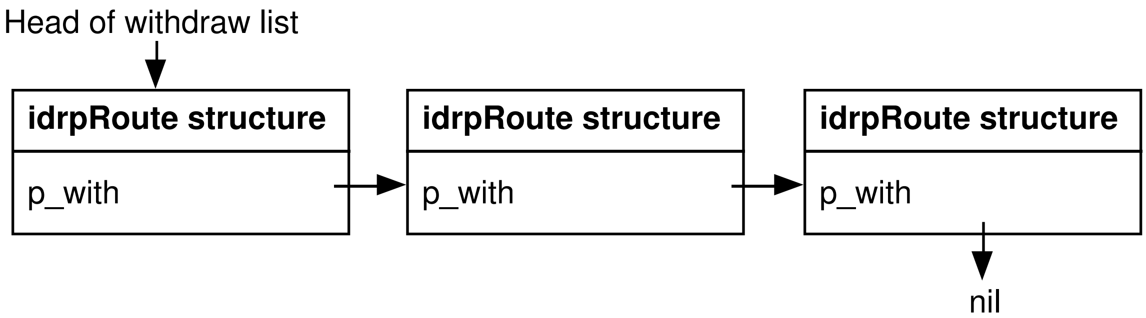


Figure 26 — Withdraw Route linked list

Parsing structures — announce list

RIB ID
pointer to next announce list structure
pointer to attribute record
Announcements Array of link structures, one entry per protocol family (ISO or IP). Link structures include head and tail, and point to idrpRoute structures.
Withdrawals Array of link structures as above.

Figure 27 — Announce list

Update parsing and processing structures

Parsing results
route ID from PDU
Withdraw structure containing count of withdrawn route IDs and array of route ID values
linked list of idrpRoute structures, containg the NLRI info from the PDU free PDU flag

Error structure (if PDU was Error PDU)
error code
error subcode
error data 1 pointer
error data 1 length
error data 2 pointer
error data 2 length

Figure 23 — Parsing Structures for Updates

Refresh PDU structures

idrpRefresh structure
pointer to next refresh structure on list (refresh list pointers are kept in the idrpPeer structure in the ‘refresh’ pointer)
pointer to update PDU that is awaiting refresh completion
parse results array for processing for this update PDU

refresh_info structure
sequence number of RIB refresh start PDU
head of idrpRefresh PDU structure
tail of idrpRefresh PDU structure
sequence number of last PDU in RIB refresh
count of PDUs to process
RIB ID

Figure 24 — Refresh PDU structures

The size of the inbound route hash table is configurable at compile time. The trade-off is (as always) between table size and search time (larger table = shorter search time).

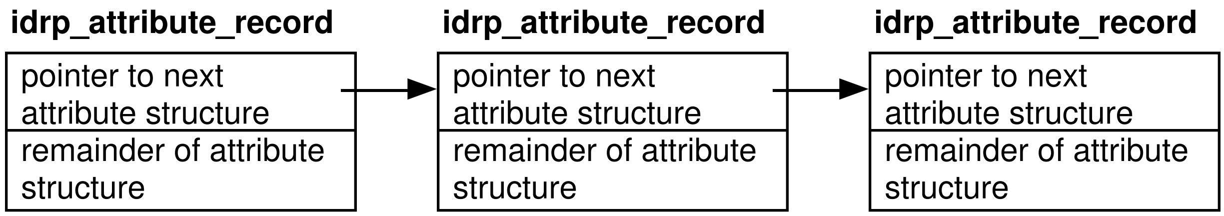
Below is a diagram of an inbound route hash table entry, keyed on route ID. The NLRI linked list is linked via the p_next_nlri field of the idrpRoute structure.

Inbound route hash table entry

route ID	pointer to next entry on chain (that is, next entry whose route ID hashed to this value	array of pointers to idrpRoute, keyed on address family

Figure 25 — Inbound Hash Table Structure

IDRP attribute linked list



Only one attribute structure is created per unique set of attributes. PDUs bearing attributes identical to an existing attribute record have their route IDs added to the route_id_list chain for that attribute record.

Figure 22 — IDRP attribute linked list

7.1.2.3. IDRP lists for IDRP route processing

The two building blocks, the idrpRoute Structure and idrp_attribute_record, are combined on lists to process withdraw route_ids and Attributes from inbound BISPDUs. The BISDPU is processed into a update structure. In turn this update structure may be included in a Refresh PDU processing structure.

If the BISDPU is valid, the route_id is added to the inbound Route_id hash table. This table links the route_id to the list of idrpRoutes for this inbound route_id from this peer.

The phase1 processing takes this structures and generates either a link list of withdrawn routes or a linked list of idrpRoutes each representing an NLRI. These lists are order by family and linked onto an announce list for further processing.

If the phase1 processing of routes requires transmitting routes to internal peers, the announce list has policy run on it. A send list is generated and handed to the output BISPDU generation routines. Each of idrpRoutes in a send list is tagged with a route_id and linked to a outbound route_id list. These circular list may be different for each peer. Therefore the pointers are stored in each idrpRoute in an array of pointers, called the Outbound Announcement Array (route_out array of structure type idrpRoute_out). The send_update_pdu routines places the resulting BISPDU into a IDRP output buffer.

IDRP attribute record structure

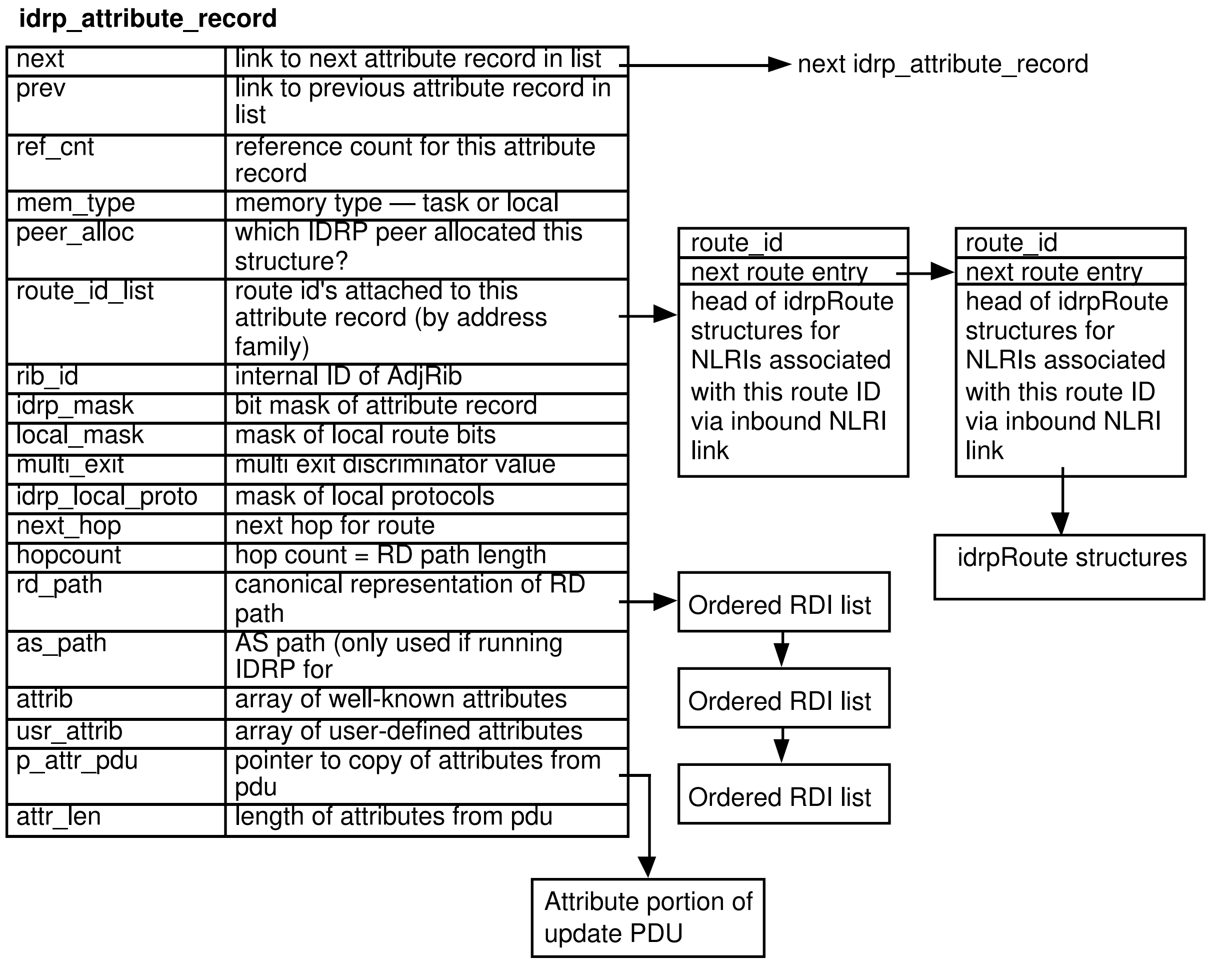


Figure 20 — IDRP attribute Record Structure

IDRP attribute array

Array entry 0 — Local RD path attribute
special attribute to encode local RD path,
which we add to the received RD path

Array entries 1-n by type (as defined in 10747)
Flag: Attribute present?
Attribute-specific flags
Length of attribute
Pointer to location of attribute in PDU

Figure 21 — IDRP attribute array

The idrp_attribute_record also keeps the RD path listed as a set of RDIs listed in "canonical" format, which is defined to be the ordering of RDIs described in the IDRP protocol specification. This ordered format is a by-product of the way the Merit IDRP code searches for duplicate RDIs in the RD_PATH attribute. It has been stored in order to aid policy calculation done on the exclusion or inclusion of an RDI or a set of RDIs. An example of these RDIs might be the exclusion of sending the non-critical traffic through an aircraft RD.

The Merit implementation of the IDRP protocol serves multiple protocols. The route_id_list contains information protocol family, such as but not limited to CLNP or IP. Each entry on the route_id_list contains a set of routes tagged by a Route_id. These routes are idrpRoute structures linked together by the p_next_nlri pointer.

In an IDRP that supports only the default RIB, the rib_id in the attribute will only be zero. IDRP for Delivery 1 (Basic IDRP, simple Policy), and Delivery 2 (Basic IDRP full Policy information) will have only zero in the rib_id.

Attributes and QOS

In a QOS environment, the "rib_id" will still remain a integer value. This integer value will be mapped to a group of distinguishing attributes. The reduction of the rib_id to a simple integer value will simply handling and comparison of IDRP routes.

For the first implementations of QOS, if all other attributes are the same and the rib_ids are different the IDRP code will create a second idrp_attribute record. IDRP attribute records will be linked for each rib_id. Zero will always be the default RIB identifier.

List of Attributes

The idrp_attribute_records are linked on a list. These lists may be scanned for to re-use an attribute list upon receiving a new BISDPDU or for policy calculations.

The first implementations of QOS will have an attribute list per rib id. Each rib id will represent a unique set of distinguishing attributes supported by the local node's gated tables. Rib id of zero will represent the default route. A table will map between the Distinguishing attributes and the rib_id. Therefore, the current idrp_attribute_list will become an array of lists.

IDRP_STATUS_ BEXT_EXT	best external route for NLRI set in phase1 processing upon add, modify of route set in phase3 upon loss of external neighbor and full route delete (real routes calculated by Best External route structure)
IDRP_STATUS_ LOC_RIB	Phase3 processing recognizes this as a gated active route
IDRP_STATUS_ MIN_ADV	Minimum route advertisement timer running
IDRP_STATUS_ MIN_ADV_CHG	Route change occurred while route had minimum route advertisement timer running
IDRP_STATUS_ WITH	Withdraw set on this route
IDRP_STATUS_ DEL_SEND	Delete this route after it has been sent to the remote peer
IDRP_STATUS_ REPLACE	Explicit Withdraw Replace route
IDRP_STATUS_ DELETE	Delete this route
IDRP_STATUS_ RECONFIGURE	This local route is subject to reconfiguration. If flag not cleared after configuration file parsing done, delete the route as no longer valid.
IDRP_STATUS_ LOCAL_NEW_ ROUTE	This IDRP route is a new local route and needs to be added to gated routing table.
IDRP_STATUS_ WITH_EARLY_ PROC	This IDRP routes withdraw was processed early because it is on an outbound list that had several ID's removed.
IDRP_STATUS_ WITH	Withdraw this IDRP route
IDRP_STATUS_ LOCAL_ROUTE	This flag is defined as the absence of all other flags after configuration and initialization finishes.

7.1.2.2. Attribute records

Attributes sent from the same peer in multiple BISPDUs may be the same. The attribute records contain a route_id_list that stores information per "route_id" about the NRLIs sent in the BISPDU. The attribute record contains processed information about a group of routes which share:

- the same next hop
- the same RDI path
- the same attributes
- multi_exit_disc value
- Route server flag

For received IDRP attribute information, the idrp_attribute_record also stores the attribute information as it was received. This storage eases the re-transmission of the byte stream for outbound routing information. The information received in the PDU is indexed by the idrp_attribute_array of pointers and byte counts for each IDRP attribute. The zero element of the this attribute array needs special mention. It is use to store the local RD added to the RD_PATH attributes. Since IDRP has no zero attribute, element zero of the array serves this function.

1) to the inbound list of NLRI's associated with a route_id.

A locally configured route associates the NLRIs with a local group of routes which do not have a route_id.

2) Processing links

Withdraw route processing link
announce route processing link

3) Timer links

links to minimum route advertisement timers, either the local RD timer or the remote RD minimum route advertisement timer.

4) AdjRib out links

An Array of pointers allow the route to be linked to the route groups (via route_id) send to Adjacent BISs.

5) Best external routes links

Links to list of all externally-learned routes to this NLRI, sorted in order of IDRP preference, such that the best external route can easily be referenced.

IDRP route links to gated tables

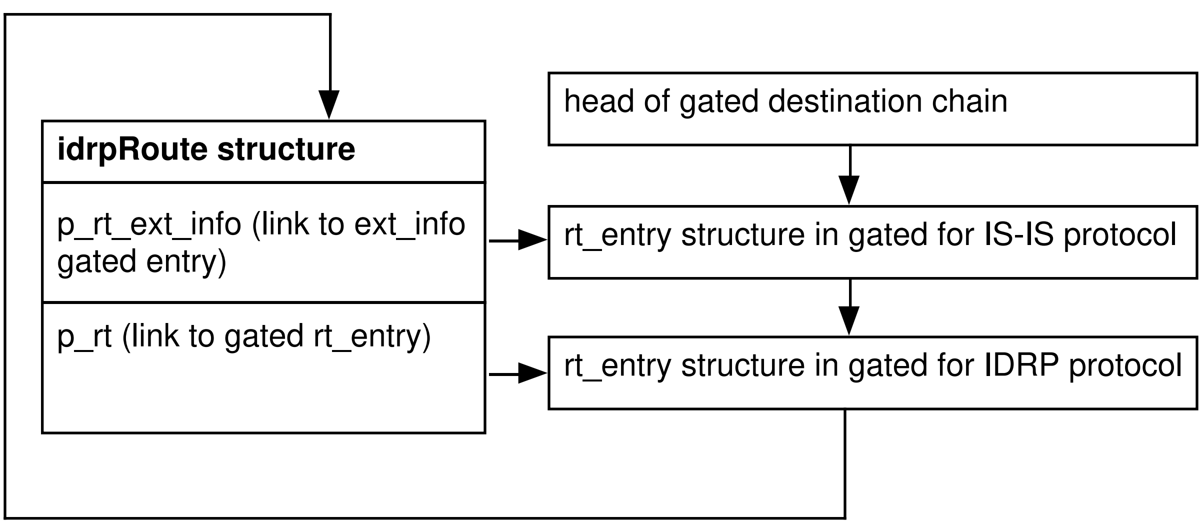


Figure 19 — IDRP route links to gated route structure

IDRP Route Status Flag	Use of Status Flag
IDRP_STATUS_ ADJ_RIB	set in a route if route added to AdjRib. set in phase1 or phase3 processing

The NRLI stored in the IDRP route is also available in the gated rt_entry structure, but is repeated here to keep the idrpRoute independent of the gated route structure.

IDRP route structure

Gated and other general use links	
rt_header	space for gated routing table stuff
p_rt	link to gated routed table
p_rt_ext_info	link to external info in gated routing table
peer	link to idrpPeer structure for peer from which we received this
p_loc_rt_peer	pointer to peer description for local route
p_attr	link to IDRP attribute record structure
Route processing links	
p_next_nlri	Next NLRI associated with inbound route ID
p_with	Next NLRI on withdrawal list
p_ann_nlri	announcing NLRI
p_min_adv	link for min advertisement timer list
p_min_advRD	link for min advertisement within RD timer list
route_out	Array of outbound route IDs for this route, indexed by peer. Also contains list of other NLRI associated with this route ID.
Best external routes links (null for internal routes)	
p_next_best	link to next-best external route
p_better	link to next-better external route
p_p_best_ext	pointer to idrp_best_ext structure. Reference off of idrp_best_ext to find the best external route.
IDRP route information	
mem_type	type of gated memory this structure was allocated from (can be local or task)
route_id_in	Route ID received from peer
pref_cal	calculated IDRP preference
pref_rcvd	preference received from internal peer
nlri	NLRI stored in prefix
family	Protocol family route belongs to (ISO or IP)
nlri_id	nlri
status	route’s status, including flags for: • Route in • Route in • Route is best external route • Min advertisement timer running for this route • Route entry reflects external info • This route is being withdrawn • Delete this route after sending it to all peers • This route changed while waiting on min advertisement timer • This route reflects a withdraw with replacement • Min RD advertisement timer running for this route • Route is being deleted • (Local) route is being reconfigured • This (local) route is new • This route should be processed early, to go with others of same route • Local route (should have no other status bits set)

Figure 18 — IDRP route Structure

The idrpRoute contains many links to aid in processing of routes into AdjRibs and into AdjRibOuts. The routes have links:

7.1.2.1. idrpRoute structure

Links to Gated Routes

The IDRP route contains data structures that allows the idrpRoute to be linked as the rt_data portion of the rt_entry of a gated route. One rt_entry exists for each protocol's route for an NLRI in the gated radix tree for a family.

For an IDRP learned route, a single link in the idrpRoute structure links it to the gated structure. In routes imported from external information (EXT_INFO routes), have two links to the gated routing table.

The first link is from the idrpRoute Structure to the gated rt_entry structure from which the gated route was imported. The second link is the gated route for created to hold the IDRP specific information. This double entry is a "short-term" structure. Long term, the gated structure will be expanded to allow a pointer to the idrpRoute (or a new structure with a subsection of the idrpRoute structure). The "double-entry" created from the external information is flagged to never be installed in the FIB since the real route is the exterior information route. Only this route should be installed in the local FIB.

Link to Attribute Record

Each idrpRoute links to an attribute record. The reference count on an attribute record indicates how man idrpRoutes are linked to a particular attribute record. idrpRoutes are linked by route_id and family to the attribute record (see the idrp_attribute_record section for details).

Links to IDRP attribute record structure have a special code in the route_id for local routes:

local_route = -1
ext_info = -2
initial set-up of route = -3

While these values are valid IDs for route_ids, the flags in the idrp_local_mask of the idrp_attribute record will have a flag that indicates local_route or ext_info.

The idrpRoute structure also contains duplicate information from the attribute record of:

- the route_id_in
- the received preference
- family of the NLRI
- hopcount

This duplication was initially added to aid debugging. However, it has provided easy reference for many routines.

The idrpRoute stores the following unique information:

- calculated preference for the route (policy calculation result for PREF)
- status of the Rout (AdjRib, LOC_RIB, best external Route, minimum route advertisement timers running, and other status)
- nlri_id that corresponds to family.

7. Description of algorithms

Section 7 describes the data structures used by the Merit IDRP implementation, and the major algorithms used by the IDRP code and gated.

Section 7.1 describes the data structures related to the Routing table. Section 7.2 describes the data structures related the IDRP neighbor (or peer). Section 7.3 describes the structures related to the Policy Information Base. Section 7.4 describes the data structures related to the Network Management Information Base. Section 7.5 contains information about what algorithms are used in route up. Section 7.6 contains reference to the Authentication algorithms.

Section 7.3 on Policy structures contains information on the rough policy structures of the Delivery 1, as well the as initial notes on the Delivery 2 policy structures. Also, to aid in configuring the current policy, a few handy examples for Delivery 1 policy will be included.

7.1. Routing table structures

7.1.1. Gated routing structures

Two types of routing table structures are used in the Merit IDRP implementation in gated, gated routing structures and IDRP specific routing structures. These routing structures are lightly linked to allow the IDRP code to be detached from the gated routing structure.

The gated routing table supports the creation of a LOC_RIB from routing information from many different protocols. The gated routes are stored in a radix tree for the protocol family (CLNP or IP). The rt_head entry links all the routes for a particular destination or in IDRP parlance NLRI. Routes received on any protocol linked to the gated rt_entry.

Like a LOC_RIB, gated only has an active route per NLRI. In fact, the LOC_RIB for the IDRP route is the list of active routes for the gated protocol.

The routing table is also threaded by a gateway.

Gated will notify tasks of changes to the active route, but it does not maintain an ability to notify tasks of changes to non-active routes. Therefore the IDRP phase1 processing, must be done within the IDRP code.

Gated has a task which installs the LOC_RIB into the forwarding table of the UNIX machine it is running on. If like AIX and BSD/386, the routing table supports both the IP and CLNP FIB with the BSD 4.4 routing table calls, then the “gated” code can install these routes into the routing table. However, if a UNIX system such as the Sun 4.x operating system does not support the CLNP protocol for insertion in the routing table the gated code cannot support the installation of OSI routes into the “FIB” of the UNIX system.

The basic building blocks of the gated routing table are not covered in detail in this section.

7.1.2. IDRP routing structures

The IDRP has two basic routing information structures: the idrpRoute and the idrp_attribute_record. An idrpRoute exists for every unique NLRI-attribute pair. An idrp_attribute records stores the attribute information either received in an BISPDU, or configured as a local route, or exported from another protocol on the gated router.

idrp_min_advRD_rt

process one minimum route advertisement timer expiring for route from this RD.

Please note that the timer routine for the local RD is only defined for the local node.

6.5. Use of the minimum route advertisement timers

Delivery 1 IDRP minimum route advertisement code has undergone very minimal testing in a 4-6 node network. The functioning of these routines do not inhibit the function of the IDRP routes.

We anticipate 2-3 months of continuous testing and refinement of the IDRP routing code. Both the logic that handles various routing table updates, and the minimum route advertisement timers require a great deal of testing:

- on a small (4-6) node test network
- on a larger test network.

Prior to deployment in the NSFNET backbone, the interaction between these minimum route advertisement timers and the various routing scenarios must tested.

This section will be updated with design comments on how to use the minimum route advertisement timer.

Routine called:

idrp_min_adv_list - adds all of announce list NLRI to appropriate minimum advertisement lists.

This routine in turn calls either:

idrp_add_min_route_adv (remote RDs) or
idrp_add_min_advRD (local RD)

These routines link the minimum route advertisement an idrpAdvRt structure. This structure has:

- link to next structure on list
- interval for timer
- routes[protocol families]

When the first minimum route advertisement structure is created, a flag is set to start the IDRP peer structure. After this initial set-up, the routes associated with a particular announce list are linked to the this idrpAdvRt structure by either the p_min_adv or p_minadv_RD pointers in the idrpRoute structure.

In the IDRP peer structure there are two blocks of the following format:

1) timer structure for gated
2) flag to indicate that IDRP route needs to have timer started
3) head of list of idrpAdvRt structure
4) tail of list of idrpAdvRt structure

The first announce list processing will set the timer structure for gated, the flag that indicates a route must be set, and set up the list pointers for the idrpAdvRtstructures which in turn contain the routes.

After all the announce list has been processed, the IDRP code starts the timers as it exits the phase3 process.

6.4. Processing the timers

When the timer expires gated calls the route to process the timer for that peer. The routine to process remote minimum route advertisement is the idrp_process_minadv. The routine to process the local RD route advertisement timer is only called by the local node task. The routine name is idrp_process_minAdvRD. Please note that we have tried to be as constant as possible with the routine names between the two timers.

The routine processing the minimum route advertisement timer takes off idrpAdvRt structures which have an interval of zero. Currently, this should only be one. However, the code is written in case we decide additional bunching of processing needs to be done. Each route in the idrpAdvRt structure is processed by walking down the chain of routes. The routes may need to be announced or deleted. The routines to handle this are:

idrp_min_adv_rt

process one minimum route advertisement timer expiring for route from another RD

6. Minimum route advertisement timers

6.1. Overview of minimum route advertisement timers

The routing fluctuations in the current Internet have been studied to see if routing traffic can be reduced. In several studies, 10%-15% of the total routes are in fluctuation at during any 15 minute interval. Some studies show that some portion of these fluctuating routes are simply oscillating rapidly between reachable and unreachable through a particular pathway. Damping out these oscillations might dramatically reduce the routing churn.

In the IDRP specification, two mechanism were insert from this real life experience to try help damp out the oscillations. These mechanism are the two types of Minimum Route Advertisement timer that exist in the IDRP protocol specification. The local RD Minimum Route Advertisement timer for routes coming from the local RD. The normal minimum route advertisement handles the routes received from other domains.

Only one of these timers is valid for any route at a time. That is, routes either come from the local RD or a remote RD.

The remote RD minimum route advertisement time tries to damp out oscillations created by other routing domains prior to passing these routes on. Each time a route is received by an External peer, and transmitted to another external peer the minimum advertisement timer is set. If the route goes away, the routing change must be passed on as bad new travels fast to inhibit blackholing of traffic. If the routes then returns, this damping function will prevent the oscillation.

The local RD minimum route advertisement timer, prevents manual configurations or exporting of routes from the Intra-Domain protocol to cause the same amount of routing fluctuations.

Let’s Try it

Out of the experience in the NSFNET, we put in these damping algorithms. Additional experience in ANSNET has proven that these damping functions are greatly needed. Curtis Villamizar (ANS) has written up ideas on weighted “route flap dampening” as an IETF draft. After additional tests, experimentation with a weighted instead of fixed timer minimum route advertisement value should be made.

The NSFNET/ANSNET is based on an flat IP address space without the hierarchical prefixes that OSI can use. Again, experimentation with real data will give use the experience necessary to set real values in for these timers.

6.2. Starting minimum route advertisement timer:

6.3. Starting minimum route advertisement timers:

After all routes are put on send_list, the PDUs are sent out to the external peers only. The phase3 route hands the announce list to the idrp_minadv_annlist.

If a route is announced out to any peer, it has a minimum route advertisement timer set. This is a optimization instead of having a timer per peer per route. This optimization still performs the necessary function which is to damp out wild oscillations.

(These additional sections will be added later)

5.7.9. idrp_local_peer_init

5.7.10. idrp_peer_alloc

5.7.11. idrp_find_peer

5.7.12. idrp_peer_update

5.7.13. idrp_config_peer_init

5.7.14. idrp_peer_update

5.7.15. local storage allocation routines

to external and internal neighbors that are connected:

- send IDRP Rib_refresh_start
- send all update PDUs packet into send list
- send idrpRibRefresh end

Note:

phase3_newpolicy calls in turn the ph3 status routines:

ph3_status_case1 - new IDRP route
ph3_status_case2 - new EXT_INFO

I believe this is adequate for the gated sequences, but I will watch to see if any other cases can occur.

Kernel routes are not imported or released, so that “route adds” will remain across reinitialization of routing routines.

5.7.8. idrp_init

Basic functions:

re-init local master task

initialize a new peer task (no bits set in the type) with timer and task allocation

Specific logic:

a) if not doing IDRP, need to delete all IDRP tasks, the master task and the peer structures. Local peer structure is a global rather than being allocated as the rest of the peer structures, so it won’t be deleted.

b) if doing IDRP,

a) re-initialize local peer (always!!)
b) Walk through peer list:

if peer has already been initialized, just wait for the peer_reinit code to do the reinitialization.

if peer has not already been initialized, then set up tasks and start timer. Start timer will allow gated to finish and stabilize before we start getting routes. A “sanity” check on gated.

b) For each connected peer, send a rib refresh sequence to refresh all routing tables

Specific functions by IDRP flag type:

DELETE: - get rid of peer structure

TRY_CONNECT - re-initialize the ETE structure

IDRP_CONNECTED - rib refresh the peer's routes

WRITE_FAILED- write failed on socket, try reinitializing the structure.

IDRP_IDLE - set in configuration, leave peer unstarted

5.7.7. idrp_newpolicy

when called: upon reconfiguration or initial stirrup

parameter(s): rtl - list of active routes

Background:

Gated has finished either a first initialization sequence or just re-initialized it's routing table and policy. The new policy is called with the current list of active routes. For the first initialization sequence, IDRP will need to create routes for any external info (EXT_INFO) it has not created routes for.

After re-initialization, a new set of active routes is handed to IDRP. IDRP can keep a connect up through a reinitialization by sending a Rib Refresh and dumping the full set of routes. Otherwise, it can drop the connection. Parameter "RibRefresh on restart" allows the RibRefresh - otherwise the connection is terminated and restarted. (This parameter - RibRefresh on restart will be implemented in Delivery 2.)

IDRP checks for the initialization case, by checking the idrp_reparsing flag. For initialization case, a simple phase3_init_status routine is called to dispatch to handle each route. The ph3_status_case2 routine which checks the flag to see if the route already has an IDRP route associated with it. If so, it calls find_idrp_route function.

Short form of logic:

If IDRP is re-initializing,

If IDRP connection up, send full routing dump via Rib Refresh on the other side with all LOC_RIB routes. This means than internal neighbors will be in sync with the LOC_RIB routes and not the best external. (We believe that this is per the specification; comments are welcome.)

Logic:

1) call phase3_newpolicy to create IDRP routes out of EXT_INFO in table, and to build an announce list of IDRP routes

2) call idrp_refresh_all_peers(p_ann_list)

5.7.4. parser.y routines

in idrp_rt_local, idrp_init_parse.c (initial and re-config)

1) if IDRP off, then idrp_init will clear out the tasks. Peers will just disappear and not give a CEASE to go down.

2) if re-parsing/initializing, local peer needs to be re-configured - routines should be the same
 a) Re-read local variables listed below
 b) reinit

3) if reparsing, then we need to locate peer structure on idrp_peer list.

 Identifiers for uniqueness:

 a) NET (always must be there)
 b) RDI (always must be there)
 c) IP address (optional with ISO)
 d) IP Interface (optional with ISO)
 e) ISO Interface - SNPA
 f) socket we are running over.
 (Can we run over multiple sockets? or have two peer structures? My guess is two peer structures)
 g) external or internal

 3.1) Other things which need to be added to the parse are the local configuration and the remote configuration parameters:

 3.2) if a peer structure is re-used, we need to re-fill the peer structure, and clear the deleted flag.

 3.3) If peer configured disabled - or current disabled function then set the flag UNCONFIGURED flag

 3.4) Policy configuration for this node.

 look for export and import
 local node configuration
 peer configurations
 policy configurations

5.7.5. idrp_reinit

This function is supposed to re-initialize the IDRP local peer structure. However, outside of configuration and initialization no other initialization is needed.

5.7.6. idrp_peer_reinit

Basic function:

 a) re-initialize each non-connected peer with ETE parameters and start to restart all non-idled peers

idrp_local_route_clean routine called later to delete any routes that are not re-configured

This is for the osilocal clause and does the same thing as the gated static routes.

4) clear the policy lists

Delivery 2 (with policy support) will free the import and export, import and aggregation lists.

Currently we have defined:

a) idrp_import_list -
A place holder. Planned to point to the list of policies threaded by NLRI for PREF function.

b) idrp_export_list
A place holder. Planned to pointer for the DIST policy structures threaded by NRLI.

c) idrp_import_paths
A place holder. Planned to point to the PREF structures threaded by attribute record.

d) idrp_export_paths
A place holder. Planned to point to the DIST policy structures thread by attribute record.

e) Aggregation functions by NLRI and
AS path will be defined for delivery 3.
gated definitions for IP aggregations will be done end of July 1993.

5) call idrp_var_init to clear global variables

Note: further information on this for Delivery 2

5.7.2. idrp_peer_cleanup

called by: task_reconfigure routine in task.c

called parameters: set flag on peer structure to delete the peer structure unless the peer configuration found in the new version of the configuration file.

5.7.3. idrp_var_inits

called by: idrp_proto_var_inits in task.c

calling parameters: none

Logic: (to be filled in later)

(Creates an attribute record to store the attribute information in. This structure is used in generic find_attr_rec routine to see if there is an attribute existing for these attributes.)

3) find_attr_rec

(Search for an attribute record with the specified attributes. If found, delete the new attribute record and use the existing attribute record. If you not found, link the new attribute record to the attribute record list.

a) idrp_free_local_att (Call this routine to free new attribute record)

b) link_local_attr_list

4) set-up the idrpRoute structure and link to attribute record

5.7. Description of routines

The following routines are described in this section:

1) idrp_cleanup
2) idrp_peer_cleanup
2) idrp_var_init
3) parser.y
4) idrp_reinit
5) idrp_peer_reinit
5) idrp_newpolicy
6) idrp_init
7) idrp_local_peer_init
8) idrp_peer_alloc
9) idrp_find_peer
10) idrp_peer_update
11) idrp_big_blk_free
12) idrp_link_peer

(More routines described by the end of Delivery 1)

5.7.1. idrp_cleanup

1) idrp_cleanup

called by: task_reconfigure routine in task.c

(gated dispatch for cleanup routines learned as tasks initialized)

parameters: none

Logic Description:

1) set the re-parsing flag
2) set delete flag in each peer
3) flags all local route structures and associated attribute record

1) for the local peer the idrpConfig Managed Objects

2) for the Adjacent BISs

1) adjacentBis Managed objects
2) extra MIB stuff from idrpPeer structure

3) the Attribute lists and associated routes

4) the local Rib

5) the AdjRibs for each peer

5.6. Local route initialization

Static IDRP local routes are configured by either the gated are configured with the “osilocal” command under the IDRP configuration. The routes are processed by the idrp_local_rt routine.

Configured External Information routes are configured via the gated “static” command, the IS-IS configuration commands or other protocol configuration commands. These routes are imported as external routes during the idrp_newpolicy routine. At this time, the static routes or protocol routes are tested to see if the IDRP code will initialize these routes.

5.6.1. IDRP configured local routes

Overview:

The idrp_local_rt is called with the NLRI, family of the NRLI, the next hop gateway for forwarding packets, and a structure of IDRP options. For Delivery 1, the route is described with the following idrp_options include:

1) rib_id
2) next_hop SNPAs
3) route server flag
4) DIST_LIST_INCL rd path
5) DIST_LIST_EXCL rd path

The idrp_local_rt tries to find this local route with NLRI, family, next hop gateway, and options. If the route already existing by calling idrp_find_local_dest routine.

If found, it clears the “IDRP_RECONFIGURE” flag. The only reason the idrp_local_rt should exists if it is a re-parsing.

If no route exists, the following routines are called to create the necessary structures for the local route:

1) idrp_create_local_gw

(creates a gated sockaddr structure to hold the next_hop information.)

2) create_local_attr_record

2) idrp_peer_terminate

The IDRP peer terminate routine:

a) releases all routes associated with peer idrp_peer_down

b) calls idrp_sm with stop_event which will shut down connection with CEASE message

5.5. Tracing change (SIGINT)

The most frequently used interrupt or initialization, SIGINT, requests a dump and the re-reading of the IDRP tracing flags. The IDRP tracing are still under development. The following tracing functions will be available, but still have to be defined:

1) Tracing of BISPDUs sent
2) Tracing of BISPDUs received
3) Tracing of IDRP state changes
4) Tracing of IDRP error messages
5) Tracing of IDRP information by neighbor.
6) Tracing of Alarm Indications for the node or by a neighbor

The IDRP alarm indications for IDRP defined by the specification are:

1) errorBISPDUsent
2) errorBISPDUconnectionclosed
3) CorruptAdjRibIn
4) PacketBomb received
5) connectRequestBisUnknown
6) EnterFSMStateMachine

Delivery 1 makes no attempt to turn off any tracing the debuggers feel is useful.

The IDRP code supports dumping the following information:

1) IDRP GDMO for the idrpConfig for the local Node
2) adjacentBis Managed Objects for each BIS neighbor
3) Loc_RIB
(See the redefinition of the ATN LOC_RIB in Section 7.4.2 MIB output structures)
4) AdjRibs
5) plus a wealth of Merit implementation variables kept.

One important addition is the printing of the attribute record lists with all associated routes.

During Delivery 1, the dump code will simply dump all known variables into the gated dump file. During Delivery 2, this dump code will be refined to allow selective dumping. If you have comments on usable tracing or dump formats, please forward these comments to the authors.

Each IDRP gated task has a reference to a dump routine. For the local peer task, the dump routine is the idrp_master_dump. For each peer task, the dump routine is the idrp_peer_dump.

The idrp_master_dump routine dumps

purpose: initialize global variables for IDRP

4) parsing routines for IDRP

called by: gated parser

purpose: parse IDRP portion of gated syntax

5) idrp_init

called by: task_proto_init in task.c

purpose: reinitialize/initialize peer and IDRP Route structures. Code looks at reparsing flag and the task assignment to peer structure to determine if a structure is new or old.

6) idrp_reinit

called by: task_reinit routine in gated

purpose: reinitialized gated master task

7) idrp_peer_reinit

called by: task_reinit for the IDRP peer tasks

purpose: handle peer status change:

DELETE peer if not configured

Reset the End to End variables so a Rib Refresh can be set if connected and reconfigured.

Disconnect peer if the peer is configured but idled.

Take a peer out of idle mode if the configuration has activated this peer.

8) idrp_newpolicy

If a peer is connected at this point, a rib refresh sequence will be sent to send new routes.

5.4. Terminate (SIGTERM)

Graceful termination of gated is signaled a kill of the process which generates a SIGTERM signal. Gated goes about gracefully closing all the tasks. IDRP tasks are the master tasks and one task per IDRP BIS neighbor.

1) idrp_terminate

The idrp_terminate routine terminates the master task, and tries to terminate any existing peers using the idrp_peer_terminate routine.

If the IDRP peer retains the connection through the re-configuration sequence, the changes to the local routes are sent to IDRP peers via either the RIB Refresh sequence or a sequence of UPDATES.

In our implementation we have chosen to allow the user make use of the power of the Rib Refresh to allow a connection to re-initialize it's routes after reconfiguration of policy without losing connections. By sending a Rib Refresh, the user can reset all the routes in the routing table. In this implementation, his reconfiguration does not generally affect the availability of a route, the connections will stay up. The RIB REFRESH sequences during reconfiguration has the power to totally reset all the Adjacent RIBs.

The IDRP specification intends that routes will remain in the forwarding base during the reception of a RIB REFRESH. Some implementations may not keep the routes up, or a user may not want to totally re-calculate the routes during a policy re-configuration. Delivery 2 of the IDRP code which implements the Policy configurations will allow the user to select whether the node will use a RIB REFRESH sequence upon re-configuration or simply send a sequence of UPDATE PDUs to transmit the changed routes.

The benefit of sending the UPDATE PDUs for small policy changes is that the routes transmitted are minimal. IDRP is an incremental protocol and does not periodically refresh all routes. The benefit of the RIB REFRESH is that it is a refresh of all routes. It will be up to the network operators to determine which function is needed at what time.

Reinitialization sequence is

1) idrp_cleanup

called by: task_reconfigure routine in task.c

(gated dispatch for cleanup routines learned as tasks initialized)

purpose:

1) clean up peers by setting delete flag in existing peers so that if the peer is not found in current configuration file it will be deleted

2) free local route list by walking through all routes associated with local attribute records and set the "IDRP_LOCAL_ROUTE_RECONFIG" flag (1st attributes in list are local attributes)

3) clean up all policy lists Note: these attributes

2) idrp_peer_cleanup

called by: task_reconfigure routine in task.c

purpose: set flag on peer structure to delete the peer structure unless the peer configuration found in the new version of the configuration file.

3) idrp_var_inits

called by: idrp_peer_cleanup

purpose: reinitialize each idrpPeer structure

notes:

1) connected peers will stay connected through policy re-init.
2) connect peers may be configured down
3) new peers may be initialized

6) idrp_newpolicy

called by: gated idrp_newpolicy routine for policy re-configuration

purpose:

1) translate any external info statically configured to IDRP routes (idrp_newpolicy)

2) run policy on existing kernel routes

3) send Rib Refresh to all connected peers

5.2.3. Route changes

As the IDRP peers are brought up, the phase 1 routes are passed via the idrp_phase1 processing. This IDRP phase1 processing sends the routes out without exiting the task.

Phase 3 route changes from IDRP or from other protocols that must be imported into IDRP are handed to IDRP for phase 3 processing by gated during its “flash” processing of routes. Each task processing the flash provide a route to process the routes. IDRP’s processing routine is the idrp_flash. “idrp_flash” in turns calls the phase3_process.

5.3. Reconfiguration (SIGHUP)

Computer networks live in a 24 hours per day, 7 day a week world. The group of routers attached to a peer may need to have a router added or deleted or the configuration variables changes. Policy for routes on a router may need to change. Gated allows this to happen while the router is active. A SIGHUP signal will signal gated to re-read it’s configuration file and re-initialize it’s routing tasks with the new peer configurations and routing policy.

The IDRP tasks reconfigure the peer structures associated with the local peer and the adjacent BIS. The new parameters for the local peer and the adjacent BIS are read from the gated configuration file.

The local routes or the Internal Systems per the IDRP specification associated with this domain are changed in the following ways:

- gated static route configuration change,
- IDRP specific route configuration change, and
- new routes configured for other local routes which IDRP imports as external routes.

The IDRP peer may either keep a connection up through the reconfiguration or terminate the connection with the local peer.

(initialize local peer variables to the default parameters)

2) idrp_peer_alloc

(allocate an IDRP adjacent BIS peer structure)

3) idrp_find_peer

(see if idrpPeer structure already exists for the this peer)

4) idrp_peer_update

(if reparsing, and found old peer structure update parameters with the new peer information.)

5) idrp_big_blk_free

(free idrpPeer structure allocated out of memory that is non-task memory)

6) idrp_link_peer

(link the peer to the idrpPeer list)

7) idrp_local_rt

(generate an idrp_local_rt. idrp_local_rt in turn calls a number of routines. See Section 5.6 on local routes.)

routines called by: gated configuration parsing routines

purpose: parsing of the IDRP configuration line options (see Section 7.2.3 as highlighted in the IDRP config syntax)

3) idrp_init

called by: task_proto_inits in task.c in gated modules table in proto_inits

purpose:

initialize the IDRP local peer structure and associated timer
initialize the IDRP neighbor BISs peer structures, and associated timers
initialize the IDRP local routes
initialize the IDRP master task
initialize the IDRP peer tasks

4) idrp_reinit - master task re-init

called by: gated re-init code for tasks

purpose: reinitialize the IDRP master task

5) idrp_peer_reinit - peer task re-init

called by: gated re-init code for tasks

5. Initialization and re-start code

5.1. Overview

Gated initialization or restart code may either start on starting up the gated daemon or upon reception of the SIGTERM or SIGHUP UNIX system. IDRP handles this reinitialization of gated to do the following things:

1) Initial program upon start-up

2) re-configure the routes or peers upon after a reconfiguration call from gated (SIGHUP)

3) Gracefully exit all IDRP sessions with peers (as gated terminates it's executing after SIGTERM)

4) Change tracing or logging on a task upon running

5.2. Initialization

5.2.1. What Init does

The initial program set-up does the following for the user:

1) reads the configuration file
2) sets up all the protocols, including IDRP with adjacent peers
3) reads in locally configured routes and routing policy from the configuration file
4) reads from the kernel the routes already configured on the router by hand
5) Computes routes to add to forwarding tables (kernel) and to send to neighbors
6) starts up routing protocols for all peers.

Initial program set-up also sets up internals of gated such as the global variables, tasks and timers to run the protocols.

5.2.2. Sequence of routines called

Gated initialization logic initialize timers, and tasks. At various points through out the gated initialization sequence it calls IDRP specific initialization routines to set-up the IDRP protocol.

Gated calls the following IDRP modules:

1) idrp_var_inits

called by: task_proto_var_inits() — routine table in proto_inits

purpose: initialize global variables for IDRP configuration parsing

2) parsing routines for IDRP

routines called:

1) idrp_local_peer_init

1) if route not announced out by IDRP, skip rest of steps

2) run distribution policy to see if announce for peer (internal aggregation is possible here, but not encouraged in IDRP specification)

[DIST(p_idrp_route) DIST - not only allows route out, but returns a modified idrp_attribute record pointer]

3) if can send route, link route on appropriate attribute record

End loop

LOOP for full list of active routes for IP

1) if route not announced out by IDRP, skip rest of steps

2) run distribution policy to see if announce for peer (internal aggregation is possible here, but not encouraged in IDRP specification) [DIST(p_idrp_route) DIST - not only allows route out, but returns a modified idrp_attribute record pointer]

3) Run aggregation specific to peer (AGGR(p_idrp_route))

4) if can send route, link route on appropriate attribute record

End loop

2) Send routes out — call Phase3_send_routes(announce_list)

a) delete idrpRoute entry
b) clear tsi bits on gated route
c) delete the rt_entry for idrpProtocol (idrp_del_rt_gated)
d) decrement reference count on the EXT_INFO information.

4.5.3. IDRP peer up routine - idrp_rt_send_init

routine: idrp_rt_send_init

parameter: peer - pointer to idrpPeer structure for peer that needs to receive full dump

Logic:

```
if (newPeer supports ISO routes)
{
get active ISO routes
}

if (newPeer supports IP routes)
{
get active IP routes
}

if (new peer == internal)
{
set external flag to FALSE
phase3_dump(active_ip_routes, active_iso_routes, external)
}

if (new peer == external)
{
set external flag to TRUE
phase3_dump(active_ip_routes, active_iso_routes, external)
}
```

4.5.4. IDRP phase 3 full routing table dump

routine: phase3_dump

parameters: p_rtl_ip, p_rt_iso

1) p_rt_ip - pointer to gated's list of active IP routes

2) p_rt_iso - pointer to gated's list of active ISO routes

3) peer - pointer to peer structure to send routes to

Logic:

1) Loop creating announce list

LOOP for full list of active routes for ISO

4.5.2.2. Phase3 send routes to external neighbors

routine: idrp_send_phase3_routes

parameter: p_ann_list - pointer to announce list

Announce list has linked list of announce structures. Each announce list structure has:

1) withdraw NLRI list
2) announce NLRI list
3) pointer to IDRP attribute record
4) rib id (zero for now)
5) null pointer to peer

Logic:

```
LOOP for each external peer:

        LOOP for each attribute record:

                Walk withdraw NLRI list
                        call send_with_attr
                end withdraw list

                walk announce list
                        call send_nlri_attr
                        (note: IDRP DIST function run on all routes
                                and policy on internal routes will
                                will let all routes pass)
                end announce list

        end of look for each attribute record

END of loop for external peers
```

4.5.2.3. Delete routes after sending the route

routine: idrp_del_phase3_routes

parameter: p_with_list - pointer to linked list of idrpRoute structures for NLRIs withdrawing.

Logic:

Loop for all of NLRIs on withdraw list:

Check for the Delete after send flag. If set in idrpRoute alone:

a) delete the idrpRoute structure (free_idrpRoute)
b) clear the tsi bits in gated route
c) let rt_delete take care of route (idrp_del_rt_gated)

if set in idrpRoute which has pointer to gated entry for ext_info:

1) do preference on new route to see if it will be announced via IDRP. If so, then create IDRP route for this external info.

subcase 1: old IDRP route going away, no new route minimum advertisement timer running

1) set min_adv_chg flag (min route advertisement will pick up change)
2) link withdrawal to announce list

subcase 2: Old route IDRP route, no new route, and no minimum advertisement timer running

subcase 3: old IDRP route, new route, minimum advertisement timer is running

1) set flag for advertisement change (MIN_ADV_CHG)
2) exit

subcase 4: old idrpRoute, new IDRP route, no minimum advertisement timer running

1) link new idrpRoute to announce list

Case 7 - deletion of active route IDRP

test: old_active route = IDRP, but no new active route. Status on IDRP route - withdrawal (?? and delete)

process the withdrawal of the route:

1) link idrpRoute to announce list as withdraw
2) if min advertisement timer is not running, set “delete after send” flag in old IDRP route
3) if min advertisement timer is running, it will handle delete of IDRP route

Case 8 - old active route = EXT_INFO route, no new active route

1) does route have idrpRoute announce bit set in gated route?

Yes, continue on with Withdraw logic
No, exit

Withdraw logic:

1) look up the IDRP Route with ext_info that matches this one (find_ext_info_rt_gated)

2) withdraw flag on IDRP route,

3) put withdraw on withdrawal list

4) if min advertisement timer running (min_adv_run set in status field of idrpRoute) then set min_adv_chg flag (min_adv_run will keep it from being deleted)

5) if the min advertisement timer is not running, (the min_adv_run flag is not set in the status field of the idrpRoute), then set the delete after send flag in the status field of idrpRoute.

3) if valid for new route create new IDRP Route

subcase 1: old route did not have IDRP route, and new route will not have IDRP route. Simply exit.

subcase 2: old route had no IDRP route (due to no announcement.) New route added.

a) link to announce list.

subcase 3: old route had IDRP route, MinRouteAdv timer is not running, new route does not generate an IDRP route

a) link old idrpRoute to announce list in withdrawal list
b) if old route has delete flagged, mark “delete after send in IDRP Route”

subcase 4: old route had IDRP route, new route does not generate an IDRP route, MinRouteAdv timer is running, Mark MIN_ADV_CHG status flag on route, Mark DELETE status flag.

subcase 5: if old route had IDRP route, and new is going to be announced and min_route_advertisement set on old route’s IDRP structure

1) set MIN_ADV_CHG in idrp_status in old route, thene exit

subcase 6: if old route had IDRP route, new route is going to be announced, and no minimum advertisement timer

a) link new route to announce list

case 5: old ext_info -> new IDRP

test: old route = non-IDRP, new route = IDRP

Logic:

1) check to see if the ext_info route has an IDRP bit set

2) if no IDRP announce bit set, simply link IDRP route to announce list and exit

3) If old route has IDRP bit, find the IDRP route (find_ext_info_rt_gated(rth))

4) if old route’s idrpRoute = this new route, then there is an error. All IDRP generated routes have no advise set on them and should not be the main route.

5) Check for the minimum advertisement timer running on IDRP route linked to ext_info route

6) If min advertisement timer running (check IDRP route status flag IDRP_STATUS_MIN_ADV_RUN), set change flag (IDRP_STATUS_MIN_ADV_CHG) and exit routine

7) if min advertisement not running, link list to the announce list.

Case 6: change of active route IDRP -> new ext_info

test: old route = IDRP, new route = ext_info (only accept IS-IS)

One route overwrites another; no deletion of old routes. we can send an implicit withdraw/change. Link changed route to announce list.

sub-case 1) Minimum route advertisement timer running

test: MIN_ADV_RUN flag set in idrp_status, MIN_ADVRD_RUN flag set in idrp_status

Logic:

1) Set flag CHG_MIN_ADV and LOC_RIB on new route. Further processing will be done when the minimum route advertisement timer expires and the route is processed

2) If withdraw and delete IDRP status flags are set in the old route, link route to withdraw portion of announce list.

sub-case 2) Minimum route advertisement timer not running, no withdraw/delete flags on the last active route.

test: IDRP status flags:

MIN_ADV_RUN and MIN_ADVRD_RUN clear
no Withdraw or Delete flags clear

logic:

1) set LOC_RIB in new route
2) clear LOC_RIB in old route
3) link new route to announce list

sub-case 3) No Minimum route advertisement timer running on the last active gated route has delete in route.

logic:

1) set LOC_RIB in new route
2) link new route to announce list
3) delete the idrpRoute linked to last active gated route
4) do an idrp_del_rt_gated on gated route.

Case 4: old ext_info -> new ext_info

test:

old route = protocol other than IDRP
new route = protocol other than IDRP

logic:

1) Do PREF on new route

if valid, flag announcement of new route
if invalid, flag no-announce new route

2) Check to see if old route has idrpRoute

1) run policy on this route to determine PREF(p_idrp_rt). If pref is zero, route will not be announced.

2) if announced, create rt_entry for IDRP so we can link in the announcements. Add link to original rt_entry into IDRP route structure.

3) rt bits set in the ext_info protocol route

4) search for existing attribute record for this ext_info route

4a) if no existing attribute record, create new attribute record
4b) if existing attribute record, link idrpRoute entry with route attributes to the ext_info route_id in the route_id_list.

5) set LOC_RIB

2) add to announce list with link via P_ANN_NLRI

Case 3) change of active route - IDRP (old_idrp_rt -> new_idrp_route)

test: old_active IDRP exists and new active IDRP for NLRI

general note:

If an IDRP route changes for a destination, the following things can have occurred:

- better route received from a peer
- withdrawal of route from a peer and selection of route from another
- withdrawal with replace (explicit or implicit) of a route from a peer which is still the best
- withdrawal with replace (explicit or implicit) of a route from a peer which changes the best route

Note:

External information such as IS-IS will come here represented as an IDRP route.

Logic:

The logic can be broken down into 3 sub-cases tested in order.

1) test for minimum route advertisement timer running.

set MIN_ROUTE_ADV_CHG_FLAG.

If old route is being deleted, but second route is being held up by minimum route advertisement timer, link withdrawal to list.

2) No minimum route advertisement timer running and original IDRP route is being deleted either by:

- withdrawal from peer, or
- withdraw of ext_info generating route

3) No minimum route advertisement timer running

case 3: old active IDRP -> active IDRP
case 4: old ext_info -> new ext_info
case 5: old ext_info -> new IDRP
case 6: old active IDRP -> active ext_info

(delete group)

case 7: old active IDRP -> no new active
case 8: old active external info -> no new active

An IDRP route is a route that comes from an IDRP peer. An ext_info route is a route from another protocol such as IS-IS. For each ext_info route announced out by IDRP an IDRP route is created and a flag set in the non-IDRP route's gated rt_entry flags.

(Note: in addition to these changes the logic for this routine must handle the minimum route advertisement processing for each route. The MIN_ADV_RUN flag is set in the idrpRoute in status field if the run has the minimum route advertisement timer running on it.)

Logic for Each case:

Case 1: new active IDRP route

test: No old active route, new active route, IDRP protocol; peer = local, internal peer, or external peer.

Logic for routine:

1) set Loc_RIB flag in IDRP route
2) link to announce list on NLRI

Note that the filtering of the route announcements in phase3 takes place as part of the announcement to each peer.

Internal peers will:

- not receive routes from other internal peers
- receive exterior routes as phase1 processing
- Local and EXT_INFO route information will be sent as part of phase3 processing.

Exterior peers will:

- receive all routes listed as active routes in phase 3

Test peers will receive an internal system peer dump.

Case 2: new active - non-IDRP protocol (such as IS-IS)

test: no old active route and new non-IDRP route

logic for routine:

1) Create external route

4.5. Phase 3 processing

Once gated has calculated the best route to any destination, it will hand the list of active routes which have changed. If a route is active and goes inactive, gated will flash the IDRP code with the idrp_flash routine. The next section describe the logic upon the IDRP code receiving this flash update.

4.5.1. IDRP flash routine

The IDRP flash routine simply calls the phase 3 processing routines.

4.5.2. Phase 3

The following is the general logic needed for the Phase 3 processing of a route. This logic is run per address family (CLNP or IP) if gated hands a change list per family.

1) walk through the gated flash list building a set of announce lists for each family. (phase3_status_change)

if (no peers)
    exit

if peers
    continue

2) send announce and withdraws to the peers (idrp_send_phase3_routes)

3) add announced routes to the minimum route advertisement lists

4) delete withdrawn NLRI idrpRoute structures which are flagged “delete after send” (DELETE_AFTER_SEND in idrpRoute status field)

4.5.2.1. Phase3 flash processing

routine: phase3_status_change

parameter: rtl_entry

rtl_entry - list of rt_entry routes received from gated in the flash update

returns: a pointer to an announce list that this code builds

Logic:

The following are possible gated status changes:

(new route group)

    case 1: new IDRP
    case 2: new active - ext_info route

(change of route group)

4.3.4. Delete external routes in Phase 1

Routes which are not in the LOC_RIB, but are the best external route must be deleted from the gated table in the phase 1 processing. Since the route is not the LOC_RIB (or gated active) route, the gated flash processing will not inform the IDRP code about the change. Routes that need to be deleted on the withdraw list are flagged with a “delete after send flag (DELETE_AFTER_SEND in idrpRoute structure status entry).

These routes are deleted only after all the withdraws have been sent to internal neighbors.

Routine: idrp_delete_phase1(list)

parameter: p_with - linked list of Withdraw NLRI/IDRP Route structures

Logic:

Walk withdraw route NLRI list doing:

if “DELETE_AFTER_SEND” is set,

1) free NLRI/idrpRoute structure from attribute record (idrp_free_nlri_att_rec)

2) re-link the IDRP output lists (idrp_free_outlist)

3) delete the idrpRoute (free_idrpRoute)

4.4. Phase 2 processing

The IDRP protocol specification calls on the Phase 2 processing to look at all the feasible routes in the Adj-Rib-Ins and determine which route:

a) has highest degree of preference of any route to the same set of destinations

b) is the only route to that destination, or

c) is selected as a results of the Phase 2 tie break rules specified in 7.16.2.1

The IDRP code in the idrp_to_gated_pref routine translates the IDRP calculated (IDRP external neighbors) or the IDRP received route (IDRP internal neighbors) to a gated route with the following algorithm:

> gated preference = IDRP preference offset + IDRP calculated/received preference *2 + IDRP tie break flag

If a gated preference matches for a destination the tie_break routine is called. This routine in turn calls the tie_break_iso routine which will follow the IDRP rules (7.16.2.1) for tie breaking.

The idrp_rt code then handles the phase 1 processing of the routes, and then adds/deletes or modifies the route in the gated routing table and goes away. The gated processing then compares the gated preference against all other IDRP preferences and all other preferences. The lowest preference is installed in the gated table. Gated interior protocols are given lower protocol preference offsets so that interior RD routes are generally preferred to exterior routes.

end loop for announce NLRIs

END LOOP for attribute record on announce list

4.3.3.8. Add withdrawals to send list for peer

routine: send_with_attr:

calling parameter: p_idrp_rt - idrpRoute structure for withdrawal NLRI

Logic:

for each withdrawal id:

1) turn withdrawals NLRIs into route IDs for this peer plus an announce list

Note: The logic here differs from phase 3. The minimum route advertisement flag is ignored (min_adv_run flag).

2) can withdrawals and announces fit in same PDU?

- yes - fit within maximum size
- no - warn and go on

3) put in withdrawals and announcements

- fit in as many NLRIs as possible
- link the NLRIs on the route_id outbound list
- if PDU full already, send the PDU (idrp_send_pdu(peer, p_send_list))
- if PDU not full but we are only allow one withdraw sequence:

4) send the PDU: (idrp_send_pdu(peer, p_send_list))

5) if PDU not full and we are allowing multiple withdrawals per PDU, add to send list.

4.3.3.9. Add NLRIs to send list for peer

routine: send_nlri_attr:

calling parameter: p_idrp_rt - idrpRoute structure for withdrawal NLRI

Logic:

1) put NLRI on send list

2) see if more space in PDU

3) if more space, exit

4) if no more space, send PDU to peer (idrp_send_update_pdu)

YES:
1) clear best external flag on current best external route
2) add best external flag on this route
3) add this route to the phase1 announce list

NO: do nothing

2) add this route to gated
(idrp_add_rt_to_gated)
(sets the RTS_NOAGE bit so that no gated timer deletes route, only IDRP processing)
}

4.3.3.6. Phase 1 - Sending best external routes to internal neighbors

At the end of processing withdrawals and announcements for external peers the following routines are called:

idrp_set_minadv_annlist(announcelist)

set min route advertisement timer on all routes that are being set to neighbors (local routes get local min Advertisement timer remote routes get global min Advertisement timer)

idrp_send_phase1_ann(announce_list) -

announce to internal peers best external route

idrp_delete_phase1(list) -

delete any routes which are deleted best external routes.

4.3.3.7. Send Phase 1 to internal neighbors

Routine Logic for: idrp_send_phase1_ann(announce_list)

LOOP doing each INTERNAL IDRP peer

LOOP for each attribute record on announce list:

Loop for all withdrawals on this entry in announce list

for each withdraw NRLI/idrpRoute structure - process withdraw into send list for a single UPDATE BISPDU. If UPDATE BISPDU is full, it will be sent to neighbor. (send_with_attr)

end loop for withdrawals

Loop for all announce NLRIs for this attribute record in announce list

process each NLRI/idrpRoute structure on announce list and add to a send list for a single UPDATE BISPDU. If the UPDATE BISPDU is full, it will be sent to neighbor. (send_nlri_attr)

based, we can compare so we can generate the best external route here.

However, if the route is not being translated into IDRP AND has a matching preference -then this code must deal with the matching preference so gated does not do tie breaking for us.

routine called: (mediate_pref_match)

The mediate preference match mediates both IDRP protocol routes and gated routes. Its return will tell whether this route should be incremented by 1 or decremented by 1.

All IDRP based routes have the following formula for gated preference:

gated IDRP offset + idrp_pref*2.

This allows the adding of one to mediate any IDRP routes easily. Hopefully, by modifying the gated route by one - the IDRP route will no longer be in contention.(This is an area for further testing)
}

if (gateway matches)
{

Gateway match means this is an implicit replace
on an external route.
1) Do error checks:
1) Check for single match on the gateway.

Should have only one match for this NLRI. If there are two, denote an error.

2) Check for null change. If attributes are the same, log but don't change.

3) Real Change denoted by any attributes changing

If a real change occurred, this route must be checked for the best_external processing. The logic is the same as the withdraw with replace route logic.

Call the common logic here:
idrp_replace_ext(p_idrp_rt, p_rt, p_ann_list, p_external, p_best_ext);
where:

p_idrp_rt	idrpRoute structure for NLRI
p_rt	rt_entry for changed route
p_ext_ann_list	announce list for phase1 external routes
p_ext	idrp_rt_chain_walk structure for external routes
p_best_ext	current best external route

return to calling routine
}

if (no gateway exists)
{
New external route has been received from this neighbor.

1) Is this route better than current best external?
(best_ext_route(p_idrp_rt, p_best_ext))

Best external route exists and is from this peer	1) set AdjRib flag in new idrpRoute 2) modify gated route to point to new idrpRoute structure, and new idrpRoute structure to point to gated route 3) unlink old route structure from route_id in attribute record, and outbound lists 4) free old idrpRoute structure 5) add new best_external route to internal neighbor announce list	1) set AdjRib and best_external route in new idrpRoute 2) modify gated route to point to new idrpRoute and new idrpRoute to point to gated route 3) unlink old route structure from route_id in attribute record and outbound lists (idrp_free_outlist) 4) free old idrpRoute structure (free_idrpRoute) 5) link new idrpRoute to internal neighbor announce list (link_ann_list)

routine name: ph1_add_ext_route

calling parameters: p_idrp_rt, p_ext_ann_list

p_idrp_route - idrpRoute (NLRI) to be added from external peer

p_ext_ann_list - best external announce list built from external routes processed in Phase 1

Actions:

1) set AdjRib flag in new idrpRoute

2) calculate IDRP preference for this route

3) calculate gated preference (Phase 2 pre-processing) (idrp_to_gated_pref(pref))

4) Look for other IDRP routes with this gateway or the same gated preference for this NLRI in gated table

Note: steps 2-4 are contained in idrp_add_route_locate

5) process based on the results of above search

```
if (preference matches)
        {

        Two types of preference matches can occur: IDRP protocol gated
        preference matches and other protocol and preferences matches.


        IDRP protocol routes will exist for all EXT_INFO (external information)
        routes that have been translated into IDRP routes in the phase3 processing
        (called during the gated flash process). So if the routes are both IDRP
```

4.3.3.5. Phase 1 - Add external route

Best External Route logic for Additions:

For each new route, the status of the route is determined by calling idrp_add_route_locate to find out the status of the new idrpRoute with this NLRI. The status of the NLRI currently in the table can be:

- No route for any external peer exists in any AdjRib (gated has no other route for this NLRI from any IDRP external peer)
- Best external Route exists in the gated table, but not from this peer.(p_best_ext return from idrp_add_route_locate points to the current best external route. A check on the peer value in the idrpRoute will indicate if it is from this peer.)
- Best external route exists and is from this peer.(if the p_best_ext returned from the idrp_add_route_locate points to this peer, then this is an implicit replace.)

The status of the new route can then be determine by using the find_best_ext routine. The table below summarizes the logic based on the status of NLRI before the route is added, and after the route is added.

NLRI status in gated table	New route is not best external route	New route is best external route
no route for any external peer exists in any AdjRib	illegal state - new route is only external	1) set best external flag in status in idrpRoute 2) add route to gated table (AdjRib) (idrp_add_rt_gated) 3) add route to internal announce list (link_ann_list)
Best external route exists in table, but not from this peer	1) set AdjRib flag in new route 2) add route to gated table	1) clear best external route flag in old best external route 2) set best external route flag in new route 3) add route to gated table (idrp_add_rt_gated) 4) add route to internal announce list (link_ann_list)

Routine looks like:

```
status = old route's IDRP status
old_idrp_route = gated routes pointer to IDRP route
switch (status)
        {
        AdjRib only:
                if (best_ext_route(p_idrp_rt, p_best_ext))
                        {
                        a) turn off best external flag on p_best_ext
                                and reset p_p_best_ext pointer to best external structure
                        b) turn on best external flag on p_idrp_rt
                        c) link to ph1 external announce list
                        (link_ann_list(p_idrp_rt, p_ext_ann_list, linked
                                via announce list )
                        }
        AdjRib & Best_ext:
        AdjRib & best_ext & LOC_RIB:
                if (best_ext_route(p_idrp_rt, p_best_ext)
                        {
                        new idrpRoute has better or equal
                        preference than old IDRP route
                         so it remains the best_external

                        a) turn on best_external in new idrp_rt
                                and reset p_p_best_ext pointer
                                to best external structure
                        b) turn off best_external in old IDRP route
                        c) link to ph1 external announce list)
                        (link_ann_list(p_idrp_rt, p_ext_ann_list,
                                        linked via announce list))

                        }
                else
                        {
                        p_best = find_best_ext(p_idrp_rt, p_ext)
                        turn on best_external in new best external
                        turn off best_external in old route
                        link new best_external to announce list
                        (link_ann_list(p_idrp_rt, p_ext_ann_list,
                                        linked via announce list))

                        }
                        break;

                } (end switch)
/* replace the IDRP route in gated route */
idrp_repl_rt_gated(p_idrp_rt, p_rt);
/* change gated route so we flash if gated route changes */
idrp_mod_rt_gated(p_idrp_rt);
}
```

Adj_RIB and best_external	1) locate all routes this NLRI and IDRP protocol keeping those that match gateway, or preference, or external. (idrp_with_route_locate) 2) if the IDRP preference of the new route less than the IDRP preference of the existing route: a) find the best external route (find_best_ext) b) if new Route still best external route, set best_external flag in new route; link into best_external chain and update idrp_best_external pointers accordingly. c) if new Route is not the best external route, set best_external flag in new best external route; relink best_external chain and update idrp_best_external pointers accordingly 3) rt_change the route in the gated table (idrp_mod_gated) 4) link best external route to send list (link_ann_list)
AdjRib, Best_external and Loc_RIB	do above Phase 3 will return either an old active IDRP -> new active IDRP or old active IDRP -> active ext_info

4.3.3.4. idrp_replace_ext

Routine name: idrp_replace_ext

called by: implicit and explicit replace of routines for external neighbors.

parameters:

p_idrp_rt - idrpRoute replacing old route

p_rt - gated rt_entry for old idrpRoute being replaced

p_ext_ann_list - IDRP announce list for phase 1 external routes to internal peers

p_ext - idrp_rt_chain_walk structure with routes of all external IDRP routes for this destination

p_best_ext - current best external route

Logic:

All External route logic except lookup for:

a) AdjRib only
b) AdjRib and Best external
c) AdjRib, BestExternal and Loc_RIB

```
                logic for this case;
                break;
        }
```

Withdraw Routes status	Action
AdjRib only	1) look up gated routes with this NLRI and IDRP protocol look for routes from same gateway, external IDRP routes, and the best_external route. (idrp_with_route_locate) 2) If new route's preference better than old best external, then: a) set old_best_external status to have best_external flag off b) set best_ext flag in new route c) modify preference so gated preference unique d) rt_change to gated tables (idrp_mod_rt_gated) e) link new route to announce list for internal peers, NLRI additions list (link_ann_list) f) link new route at head of best_external chain for NLRI; reset idrp_best_external pointer for NLRI to point to new route. 3) if new route's preference less than old best external route, a) modify preference so gated preference is unique b) rt_change to gated tables (idrp_mod_rt_gated) c) insert new route into best_external chain in order of preference

AdjRib & best external & Loc_RIB	1) go a rt_locate on this NLRI and idrpProtocol (idrp_with_route_locate) a) if gateway match - 2nd route this destination and this gateway - it's illegal 2) try to find new best_external route from other external routes (find_best_ext) 3) if new best external found, a) set best_external flag in new route; reset route's idrp_best_external structure to point to it; relink best_external list to remove route being deleted. b) Withdraw should be set in old route c) link new route to internal announce list as announce (link_ann_list) d) set delete on IDRP route e) rt_delete on old route and let gated recalculate new Loc_RIB (idrp_del_rt_gated) 4) if no new best external found, (this was only external route) a) Withdraw should be set in withdraw route b) link withdraw route to internal announce list as withdraw (link_ann_list) c) do rt_delete on this route d) free route's idrp_best_external structure

4.3.3.3. Phase 1 - Withdraw external route with replacement

routine: ph1_with_repl_route

parameters:

p_idrp_route - pointer to idrpRoute structure

p_ext_ann_list - pointer to external Announce list

External Route logic for Explicit Withdraw with Replace:

The action upon receiving a external route which is a withdraw with an explicit replace depends on the status of the route being withdrawn and replaced. The tables below describe the logic based on the status of the route. The actual routine logic looks like the following:

```
switch(status of idrpRoute being withdraw)
        {
        case AdjRib only:
                logic for AdjRib only route;
                break;
        case AdjRib & best-external:
                logic for AdjRib and Best External;
                break;
        case AdjRib, Best-external & Loc_RIB:
```

Logic for Withdraw External Route (NLRI):

status of route being withdrawn	Action
AdjRib only	1) remove NLRI from attribute record by: a) pulling from route_id_list for this route_id and this peer, b) decrementing the attribute record reference count (idrp_free_nlri_att_rec) c) no outbound routes should be linked d) clear gated route of idrpRoute structure e) delete idrpRoute Structure (free_idrpRoute) f) delete gated route entry with rt_delete (idrp_del_rt_gated)
AdjRib and best_external	1) search the gated table for a gateway match, idrpPreference match, and a list of external routes (rt_with_route_locate) a) if gateway match - illegal unless min_route_advertisement set on route 2) find new best external route (find_best_ext) 3) if new best external exists: a) set best_external flag in new route; reset route’s idrp_best_external structure to point to it; relink best_external list to remove route being deleted. b) link new route to announce list for internal peers (link_ann_list) c) unlink NLRI from route_id list d) unlink route from outbound route lists (idrp_free_outlist) e) unlink from attribute record f) delete old idrpRoute structure (free_idrpRoute) 4) if new best external route does not exist: a) link withdraw NLRI to announce list for peers (link_ann_list) b) set delete after send flag in idrpRoute c) free route’s idrp_best_external structure

One linked exist exists per family of addresses. For example, there will be one linked list for all ISO addresses. Each linked list consists of idrpRoute structures. An idrpRoute is created for each NLRI received from a peer. These idrpRoute structures are linked via the p_with pointer in the idrpRoute structure.

2) inbound linked list of routes to be added linked with the p_next_nlri in idrpRoute Structure

3) a pointer to Attribute record based on the inbound Route_id of the route. The route_id_list structure points to the first NLRI on the chain of NLRI for this route_id.

4) peer that these routes came from

As noted earlier, the additions list may contain:

- NLRIs with withdraw with replace found in PDU
- NLRIs which are implicit withdrawals
- NLRIs which represent new routes.

These route additions are added to the gated tables much as the internal routes are. However, the phase 1 process to determine best external route must be done on external peers.

Logic for phase1_external routine:

```
Loop for each NLRI family {
        loop handling withdraw NLRI list of idrpRoute Structures
        linked by p_with
                ph1_with_ext_route called to process each route
        loop end
        loop handling announce NLRI list of idrpRoute structures
        linked by p_ann_nlri
                 if (withdraw/replace)

                        call ph1_with_repl_route - to process
                                each route with withdraw/replace
                else
                        call ph1_add_ext_route
                end if
        loop end
NLRI family loop end
send best external routes to internal neighbors
        (send_best_ext)
delete routes that are only best external
        (del)routes_best_ext)
```

4.3.3.2. Phase 1 - Withdraw external route

routine: ph1_with_ext_route

parameters:

1) p_idrp_route - idrpRoute withdrawing

2) p_ann_list - announce list

Note: Recent gated changes to grab the destination address structure and mask structure out of fast allocated memory will require a change to this routine.

4.3.3. Phase 1 - Routes from external neighbors

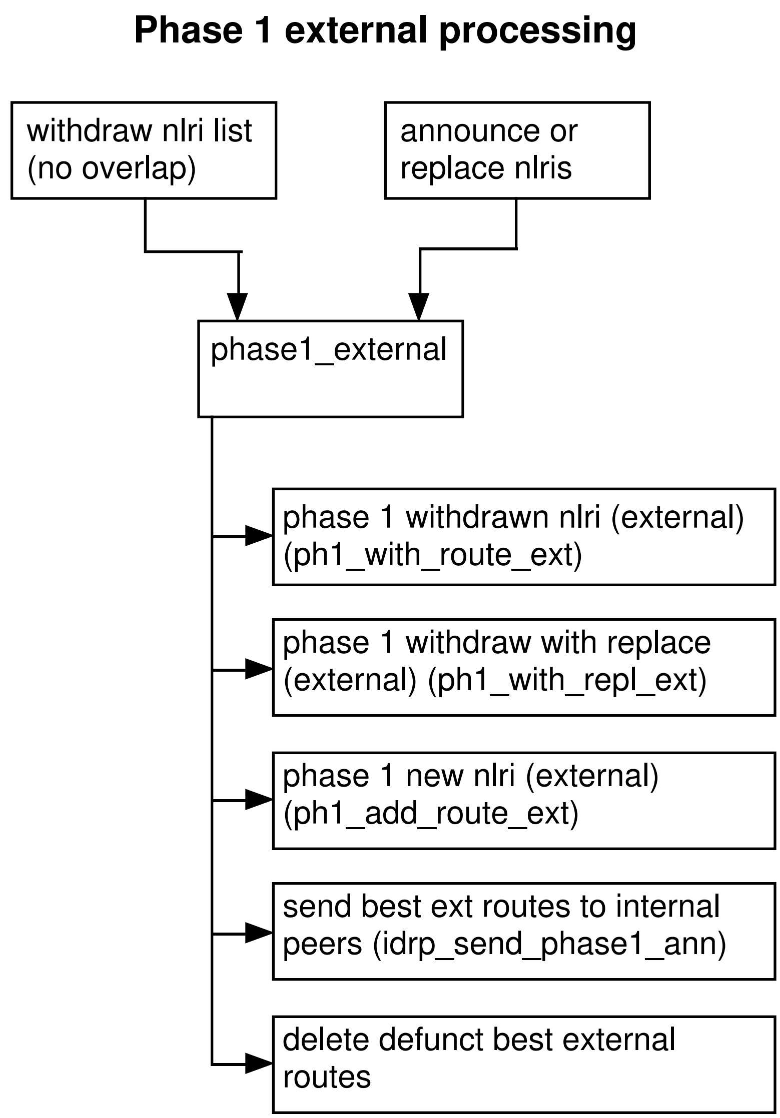


Figure 17 — Phase 1 External Route Processing

4.3.3.1. phase1 external routes

Routine: phase1_ext

parameters passed:

p_ann - announce list which contains:

1) Inbound linked list of withdrawn routes

4.3.2.8. free_idrpRoute

routine: free_idrpRoute

calling parameter: p_idrp_rt

Logic:

1) Check that all p_idrp_rt references have been released (idrp_attribute_record, route_out id lists)

2) use task_mem_free to free the idrpRoute structure

4.3.2.9. idrp_del_rt_gated

routine: idrp_del_rt_gated

calling parameter: p_rt - pointer to gated route

Logic:

1) do an rt_delete on the rt_entry structure (gated route)

4.3.2.10. idrp_add_rt_to_gated

routine: idrp_add_rt_to_gated

calling parameter: p_dest, p_mask, p_idrp_rt

Logic:

1) create a rt_params block

2) fill in rt_params block from p_dest, p_mask, and p_idrp_rt structure information

3) call gated routine rt_add

Note: Recent gated changes to grab the destination address structure and mask structure out of fast allocated memory will required a change to this routine.

4.3.2.11. idrp_mod_rt_gated

routine: idrp_mod_rt_gated

parameter: p_dest, p_mask, p_idrp_rt

Logic:

1) generate parameters for rt_change call such as gated preference

2) call gated routine rt_change

4.3.2.5. find_best_ext

routine: find_best_ext

parameter: p_idrp_rt - pointer to route we want to add.

Logic:

Just return p_idrp_rt->p_p_best_ext->p_best_ext. This returns the (pre-computed) best external route. See figure 32 for further clarification.

4.3.2.6. idrp_free_nlri_att_rec

routine: idrp_free_nlri_att_rec

calling parameters: p_idrp_rt, with

p_idrp_rt - pointers to the idrpRoute structure to free from attribute record

with - Withdrawal flag indicates that route is linked through the p_with indicator instead of the p_next_nlri pointer.

Logic:

1) get the family of the route

2) find the idrpRoute in the route_id chain in IDRP attribute record (idrp_find_routeid_chain)

3) look for NLRI inside the route_id chain

If you find a match to the pointer to the route, re-link it. If with flag is set, use the p_with to re-link it. If the with flag is not set, use the p_next_nlri flag.

4) if Route_id list is empty, free it

5) if idrp_attribute record reference count goes to zero, free it.

4.3.2.7. idrp_free_outlist

routine: idrp_free_outlist

parameters: p_idrp_rt

p_idrp_rt - points to the idrpRoute structure for route

Logic:

```
Loop for n peers configured
        for route_out entry, relink circular list
End of loop
```

```
 (get list of routes for this NLRI and IDRP protocol)
if (no gated route entries)
        {
        no gateway match
        no preference match
        }
else
        {
        for all routes for this NLRI and IDRP protocol
        do the following:

        look for equal pref or gateway match
        if (gateway match)
                        -> save as matched gateway)
        else if (pref_match && not_gateway)
                        -> save list of preference matches
                         on idrpRoute_list
        }
```

4.3.2.4. idrp_with_route_locate

routine: idrp_with_route_locate

calling parameters: p_idrp_route, p_gated, p_pref, p_ext, p_best_ext

1) p_idrp_route - Pointer to IDRP Route

2) p_gate - pointer to idrp_rt_chain_walk structure for keeping same gateway information

3) p_pref - pointer to idrp_rt_chain_walk structure for keeping same preference in IDRP information

4) p_ext - pointer to idrp_rt_chain_walk structure for keeping all other external routes for this destination

5) p_best_ext - pointer to idrpRoute structure for best external route

Logic:

1) get the beginnings of the gated route list for this destination

p_idrp_rt->p_rt points to rt_entry for this destination for IDRP protocol for this gateway. Use rt_entry's pointer to head to get linked list for this destination.

2) walk down the chain of routes looking for routes with IDRP protocol set.

1) if find same gateway, link to that list
2) if find same preference, link to that list
3) if find external, link to that list
4) if find best external, save pointer

return these pointers to the calling routine.

6) p_ext - pointer to idrp_rt_chain_walk structure for keeping all other external routes for this destination

7) p_best_ext - pointer to idrpRoute structure for best external route

Action:

Calculate idrpPreference on the new route.

4.3.2.2. PREF

routine: PREF(p_idrp_rt)

calling parameter: pointer to idrpRoute structure of NLRI

Actions:

1) calculate preference given policy

2) log it to Network Management (gated log file) if the calculated preference does not match received preference.

3) calculate gated preference on the new route

4.3.2.3. idrp_to_gated_pref

routine: idrp_to_gated_pref(p_idrp_rt)

calling parameter: pointer to idrpRoute structure of NLRI

Actions:

1) Use received preference to calculate gated Preference. Formula is:

gated preference = idrp_preference_offset + idrp_received_preference * 2

2) locate any route from same gateway or preference
  a) same gateway - should be illegal
  b) same preference - tie break in IDRP code
  c) any external routes from IDRP
  d) best external route from IDRP

3) Walk the gated route list down from the gated entry pointed to by this destination and the IDRP protocol. Link any gateway that are the same as this route to the idrp_rt_chain_walk structure for gateway. Link any IDRP routes with the same preference as this route to the preference idrp_rt_chain_walk structure for this gateway.

Link any external peer route to the external idrp_rt_chain_walk structure for external peers. Save the best external IDRP route in p_best_ext_rt pointer.

Algorithm for search:

p_rt = rt_locate(state, dest, mask, protocol);

```
                    6) issue a rt_change to gated
                            (idrp_mod_rt_gated)
                    }
else
                    {
                    /* new route */
                     add the new route to gated
                            (idrp_add_rt_to_gated)
                    }
```

Optimization note: In future optimization of code path, we may want to additionally reduce the rt_change calls made to gated if the route is not going to change its gated preference.

4.3.2. Utility routines for Phase 1 and Phase 3

The following routines are shared by gated phase one processing:

1) idrp_add_route_locate
2) idrp_with_route_locate
3) idrp_find_best_ext
4) idrp_free_nlri_att_rec
5) idrp_free_outlist
6) free_idrpRoute
7) idrp_del_rt_gated
8) idrp_add_rt_to_gated
9) idrp_mod_rt_gated
10) send_with_attr (see routine description in section 4.3.3.2)
11) send_nlri_attr (see routine description in section 4.3.3.3)
12) link_ann_list
13) link_send_list
14) send_update_reset_send
15) flush_att_send_list
16) idrp_send_list_room
17) idrp_del_routes_sent

[add more routines here over the weekend]

4.3.2.1. idrp_add_route_locate

routine called: idrp_add_route_locate

parameters: p_dest, p_mask, p_idrp_rt, p_gate, p_pref, p_ext, p_best_ext

1) p_dest - pointer to sockaddr_un structure for gated style destination

2) p_mask - pointer to sockaddr_un structure for gated-style destination mask

3) p_idrp_route - Pointer to IDRP Route

4) p_gate - pointer to idrp_rt_chain_walk structure for keeping same gateway information

5) p_pref - pointer to idrp_rt_chain_walk structure for keeping same preference in IDRP information

The gated preference comes from the IDRP preference. We need to insure that gated will not receive two routes with an equal preference. Gated will do it's own selection if we send two routes in with the same preference. Therefore, the IDRP code needs to do it's own tie breaking between any idrpRoute structures. IDRP route structures can be generated either from external routes (routes from IS-IS or other protocol routes including statically generated) or from IDRP received routes.

4) do an rt_change to modify the gated preference

(idrp_mod_rt_gated)

5) clear REPL flag

4.3.1.3. Phase 1 processing for route additions:

Routine: ph1_add_route_int

calling parameter:

p_idrp_rt - idrpRoute

Actions:

(steps 2-4 are contained in idrp_add_route_locate)

1) set AdjRib status flag in new idrpRoute

2) calculate IDRP preference for this route.

3) calculate gated preference (Phase 2 pre-processing) (idrp_to_gated_pref(pref))

4) Look for other IDRP routes with this gateway or the same gated preference for this NLRI in gated table

5) process based on the results of above search:

```
if (preference_matches)
        {
        tie_break between all the routes whose preference matches
        the new one (all same NLRI)
        }
if (gateway matches)
        {
        /* implicit withdraw/replace */
        (idrp_with_repl_int)
        1) change rt_entry gated pointer to new idrpRoute
        2) change idrpRoute pointers to new rt_entry
        3) relink around it in route_id chain
        4) free from any output list
        5) get rid of the this idrpRoute by unlinking
            from attribute list
            (idrp_free_nlri_att_rec)
```

Status of Route being withdrawn	Action
AdjRib only	1) free this NLRI from the attribute structure (idrp_free_nlri_att_rec) 2) remove this NLRI from any outbound lists (idrp_free_outlist) 3) free the idrpRoute structure and set the gated route pointer to null (rt_entry.rt_data = NULL) (free_idrpRoute) 4) Set DELETE flag in IDRP status flags, call rt_delete so gated will delete route (idrp_delete_rt_from_gated)
Loc_RIB and AdjRib	1) set delete in status flags 2) do gated delete on route (rt_delete) and let gated flash update tell us about the change in phase 3 (idrp_delete_rt_from_gated)
best external route	illegal for internal updates
min_adv_run	illegal for internal updates
idrp_chg_min_adv	illegal for internal updates
delete after send	illegal for internal updates
status	where route deleted
AdjRib only	in ph1_with_int routine
AdjRib and Loc_RIB	in phase3_status_change routine
all other flags	illegal status

4.3.1.2. Phase 1 explicit withdraw with replace

Routine: ph1_with_repl_int

calling parameter:

p_idrp_rt - idrpRoute structure of NLRI withdraw with replace

Actions:

(steps 1-3 contained in idrp_add_route_locate)

1) calculate idrpPreference on the new route

2) calculate gated preference on the new route

3) locate any route from same gateway or preference

a) same gateway - should be illegal
b) same preference - tie break in IDRP code

Note:

Phase 1 internal processing

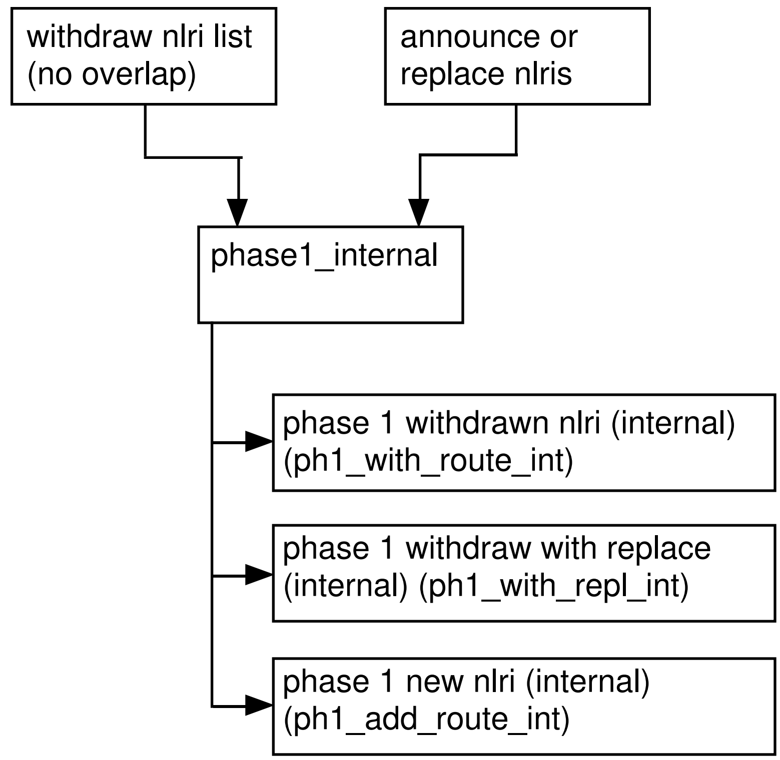


Figure 16 — Phase1 internal processing

4.3.1.1. Phase 1 withdrawal logic for internal peer

routine name: ph1_with_int_route

calling parameter:

p_idrp_rt - idrpRoute structure for NLRI removed

The ph1_with_int action depends on the status of the idrpRoute being withdrawn:

1) Withdrawal of NLRI
2) withdraw/replace
3) implicit withdraw
4) new route

The announce list passed to the phase one processing has:

1) Withdrawal list

A linked list of NLRI's to be withdrawn. These NLRIs are linked via the p_with pointer in the idrpRoute structure.

2) Announce list

This linked list of NLRI's are linked via the p_next_nlri parameter in the idrpRoute structure. In the announce list the idrpRoutes may be a withdraw with replace. In that case, the route will be marked with Replace. For a implicit withdraw with replace route, no bits will be set in the status field.

When phase1 code looks up the route in the gated table, the route may exist, or not exist from this peer. If the route exists from this peer, the route is an "implicit withdraw". If the route does not exist from this peer, it is a new route.

3) pointer to the attribute record

The attribute record structure created from the UPDATE BISPDU is passed to the phase1_internal routine. A part of this structure is the list of idrpRoute structures announced by this peer.

4) pointer to peer structure these routes came from.

The phase 1 internal processing loops processing the withdraw NLRI list. It then processes the additions into: explicit withdraw replace, implicit withdraw replace, and new route.

Overlapping of Routes:

First version processing:

In the first release of the Merit IDRP code, when routes overlap, both routes are installed into the routing tables. Since there are no policy filters in phase 1, all routes are accepted.

Policy and Overlapping routes

When Policy is enacted into the phase 1 and phase 2 processing, the overlapping routes installed will be dictated by policy. Gated's basic route look ups into the radix will need to be modified to return best match.

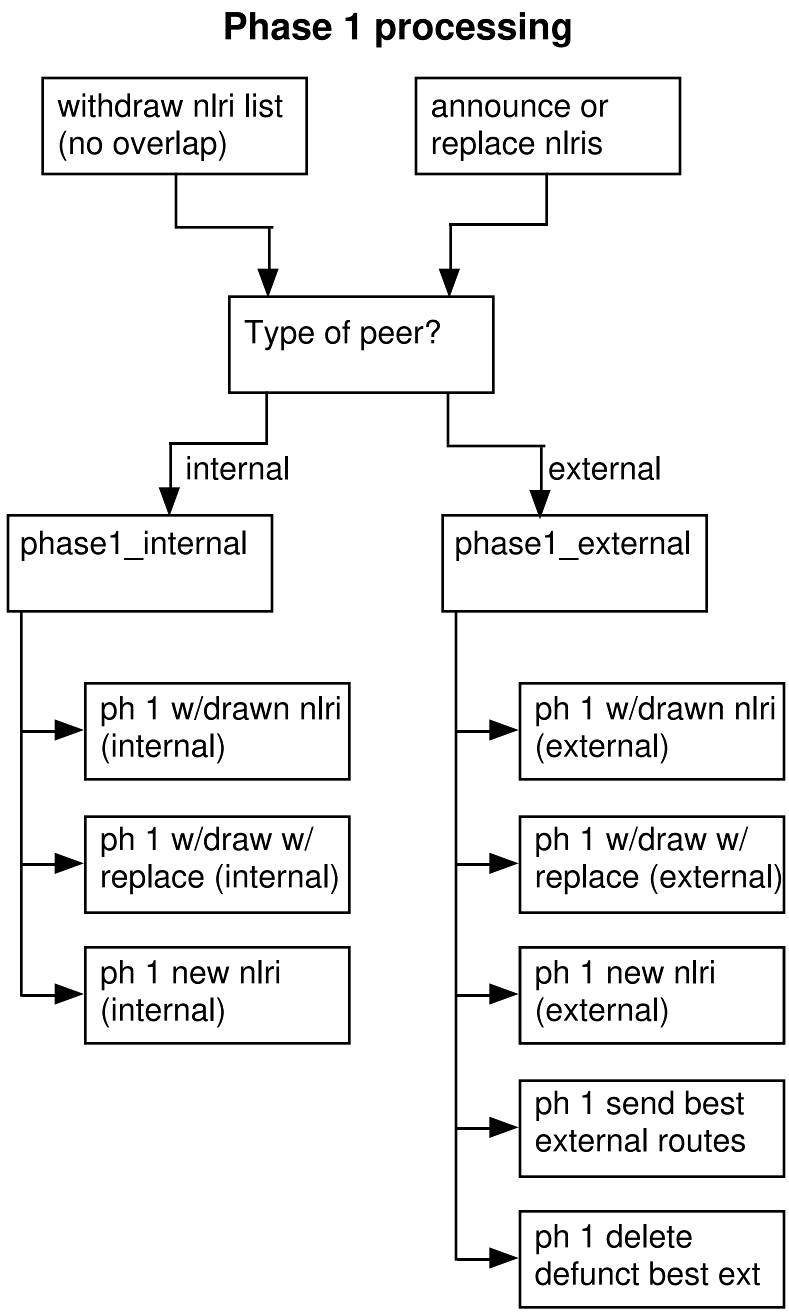


Figure 15 — Phase 1 processing of routes

4.3.1. Phase 1 processing for internal peer

Routine name: phase1_internal

Logic:

The three type of functions are processed for the IDRP internal peer: withdrawal of an NLRI, withdraw/replace of an NLRI, and a new addition of a new NLRI. Each addition may either be an entirely new route or an implicit addition. Therefore the four cases of actions are:

The phase 1 processing branches immediately depending on whether the route came from an internal peer or an external peer. Both the internal and external phase 1 processing modifies the gated tables which provide the structure for the Adjacency Ribs. In addition, the external peer processing of routes must include:

- re-calculation of the best_external route if needed,
- transmission of the best_external routes to internal peers as the IDRP phase 1 processing specifies, and
- removal of withdraw best_external route structures which are not also best route, that is in the Loc_RIB structure.

The best external route is tracked by a best external route structure created each time a new NLRI (destination) is seen by IDRP. As additional routes for the same NLRI (destination) are seen, the additional routes are linked into the NLRI structure. See figure 32 for a depection of this scheme.

In removing a route, this code deletes the idrpRoute structure associated with the gated route, and then calls the gated rt_delete function to delete the gated structure. The deletion of this type of best route occurs here because gated will not indicate these changes in the idrp_flash since these changes do not effect the active route (the Loc_RIB route).

Logic:

```
for (each withdraw NLRI in linked list)
        The NLRI list is linked via the p_next_nlri in.
        walk list of announce NLRIs looking for match
        (announce list is linked via the p_next_nlri)

        if (match)
                {
                do Withdraw replace processing
                }
        else
                {
                no match - link to chain of withdraw NLRIs
                via the p_with in idrpRoute
                }
```

Withdraw replace processing:

1) set Replace flag in status variable of announced idrpRoute for NLRI

2) rt_entry from old idrpRoute set in new idrpRoute

3) gated rt_entry points to new route structure

4) unlink old route from attribute record

   a) pull NLRI from route_id_list for route ID and peer
   b) decrement reference count of attribute record
   c) if reference count for attribute record is zero, free it (idrp_free_nlri_att_rec)

5) outbound chains for old idrpRoute relinked to exclude the route structure (idrp_free_outlist)

6) free the withdrawn idrpRoute structure created in the parse update processing.
(free_idrpRoute)

4.3. Phase 1 processing

Phase 1 processing dispatches on whether the routes came from an internal or an external Peer. The phase one processing is passed a list of NLRI to be withdrawn and a list of announced NLRIs. However, in this list there are four types of routes:

1) explicit withdraw routes (on withdraw NLRI list)
2) withdraw with replace (on announce NLRI list)
3) implicit withdraw (on announce NLRI list)
4) new route (on announce NLRI list)

The BISPDU processing up to this point has determined whether the route is an explicit withdraw or a withdraw with replace (replace set in the route added). However the other additions can either be a implicit withdraw from a peer that over writes a previous route sent by peer or a new route. Only the look-up in the current AdjRibs in the gated routing structure determines which of the last two a route is.

PDU. If NLRI is a withdraw/replace case, the Replace flag is set in the announcement list, and the withdraw NLRI is deleted from the withdraw list.

3) A list of announcements is passed to the phase 1 processing. The announcement list has:

- a withdrawn NLRI list,
- an announced route NLRI list,
- a pointer to the attribute record for these NRLIs.

The specifics of creating a withdraw list in idrp_process_pdu_routes are:

1) look up Withdraw Route ID in inbound route hash table for the peer received from.

2) if no withdraw ID exists, trace but ignore

3) if withdrawal ID exists, do steps 1-4 for each route ID

> 1) pull hash list entry. Hash entry will have route ID and a linked NLRI lists per protocol supported. Delete hash table entry for route ID

do steps 2-4 per NLRI chain (chain of idrpRoute structures) in hash list entry

> 2) walk the NLRI chain, unlinking the idrpRoutes from the inbound route_id and linking them to the withdraw NLRI list.
>
> > a) clear the p_next_nlri list
> > b) link to withdraw list using p_with pointer in idrpRoute
> > c) set flag on route to Withdraw
> > d) set status in route_id list entry in attribute record to delete
>
> 3) Remove any announced IDs from withdraw route id

This double checking removes any withdraw replace routes which might be found in the case where the withdraw id points to 10 routes to withdraw, but 9 routes are then included in the NLRI announcement.

> 4) If NLRI is not found in announce list, the withdraw NLRI structure is linked on the announce list withdraw NLRI list.

note: each withdrawal NLRI has route_id, p_with set, and p_attr still set.

4.2.5. remove_ann_dup

routine called: remove_ann_dup(p_withdraw, p_announce)

calling parameters:

p_withdraw - pointer to list of idrpRoutes that you are withdrawing

p_announce - announce list

note: remove_ann_dup called for each type of NLRI - ISO or IP

4.2.4. idrp_process_pdu_routes

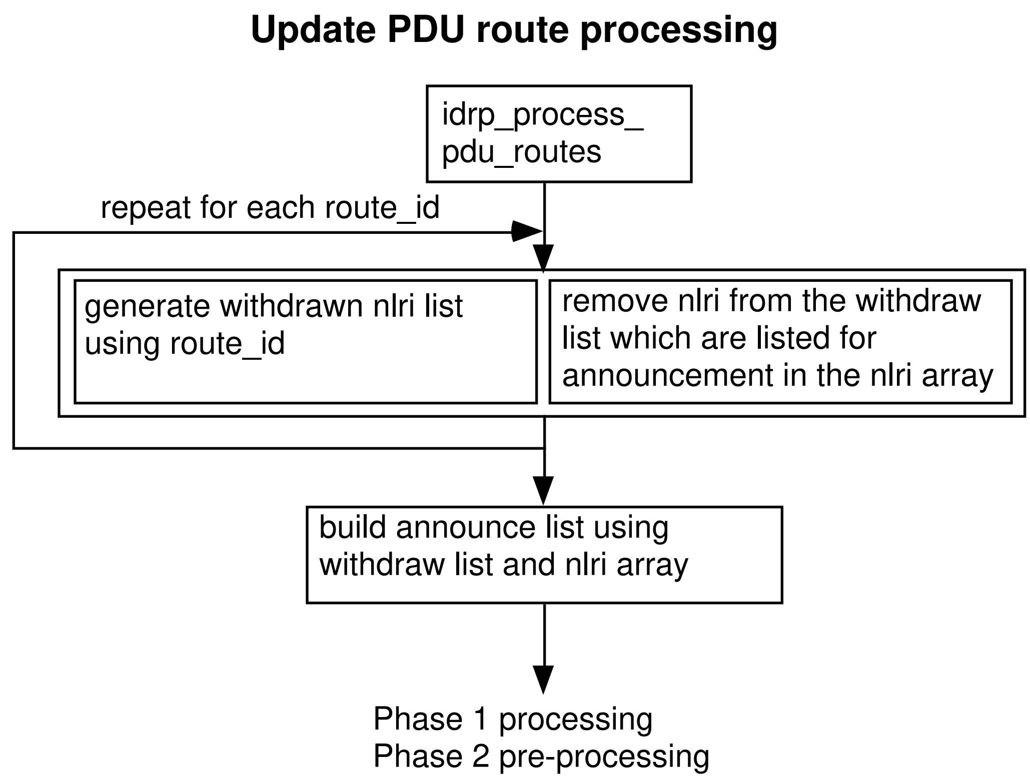


Figure 14 — Update PDU route processing

Routine: idrp_process_pdu_routes

The idrp_process_pdu_routes is called when a valid update PDU is to be processed by the IDRP code. This processing can occur as part of the normal update procedure or as part of the Rib Refresh code.

The idrp_process_pdu_routes code is handed a parsing results structure from the idrp_process_update code. This parsing results structure has a pointer to the attribute record, array of NLRI lists by family, withdraw id array and free PDU flag.

Logic:

The idrp_process_pdu routes processes the Withdraw Route IDs and NLRI announcements into a list of withdraw NLRIs, a list of NLRIs to add or withdraw/replace.

It does this by:

1) Creating withdraw NLRI list by walking through Withdraw Route IDs and looking up in the hash table idrpRoute lists for that route_id. A Withdraw flag is set in the status field of the idrpRoute associated with that destination and peer.

2) As each withdraw NLRI is gotten from Withdraw Route IDs, check it make sure it was not a withdraw/replace case where the Route Id was withdrawn but the NLRI was announced in the

If attributes exist in the update PDU, an attribute record structure is created. Attributes are parsed into this structure, and validated. If an identical attribute record is found, that attribute record is used instead of this one.

The network layer reachability information (NRLIs) are parsed out of the update PDU, and put into idrpRoute structures. These idrpRoute structures are then linked onto an inbound route_id list using the p_next_nlri pointer in the idrpRoute structure.

Each NLRI is linked on a list per family. The attribute record keeps the list of routes attached to it by idrpRoute_entry list. This structure attached to the attribute record stores the route ID of the IDRP route, and any NRLI associated with that route_id for that peer.

The reference count of the attribute record is incremented once per NLRI. Each NLRI is represented by an idrpRoute tied to a gated route structure. The parse results also contain a flag on whether the inbound PDU memory can be released to gated or not.

4.2.3. parse_update_cleanup

Routine: parse_update_cleanup(results)

calling parameter: results

results - pointer to parse_results structure

The parse results structure contains:

1) route ID - from UPDATE BISPDU
2) pointer to attribute record - created from UPDATE
3) linked list of NLRIs by family
4) flag to free PDU storage
5) withdraw structure hold withdraw IDs and count

Actions:

1) release withdraw array space
2) free the attribute record (idrp_free_att_rec)
   if IDRP_INIT_ROUTE_ID is set then no NLRI are attached: just free the attribute record
   if NLRI released, then decrement the reference count once per NLRI.
3) Decrement the Best External Route Structure reference count
4) free the Best external route structure if reference count is zero

4.2.2. parse update_pdu

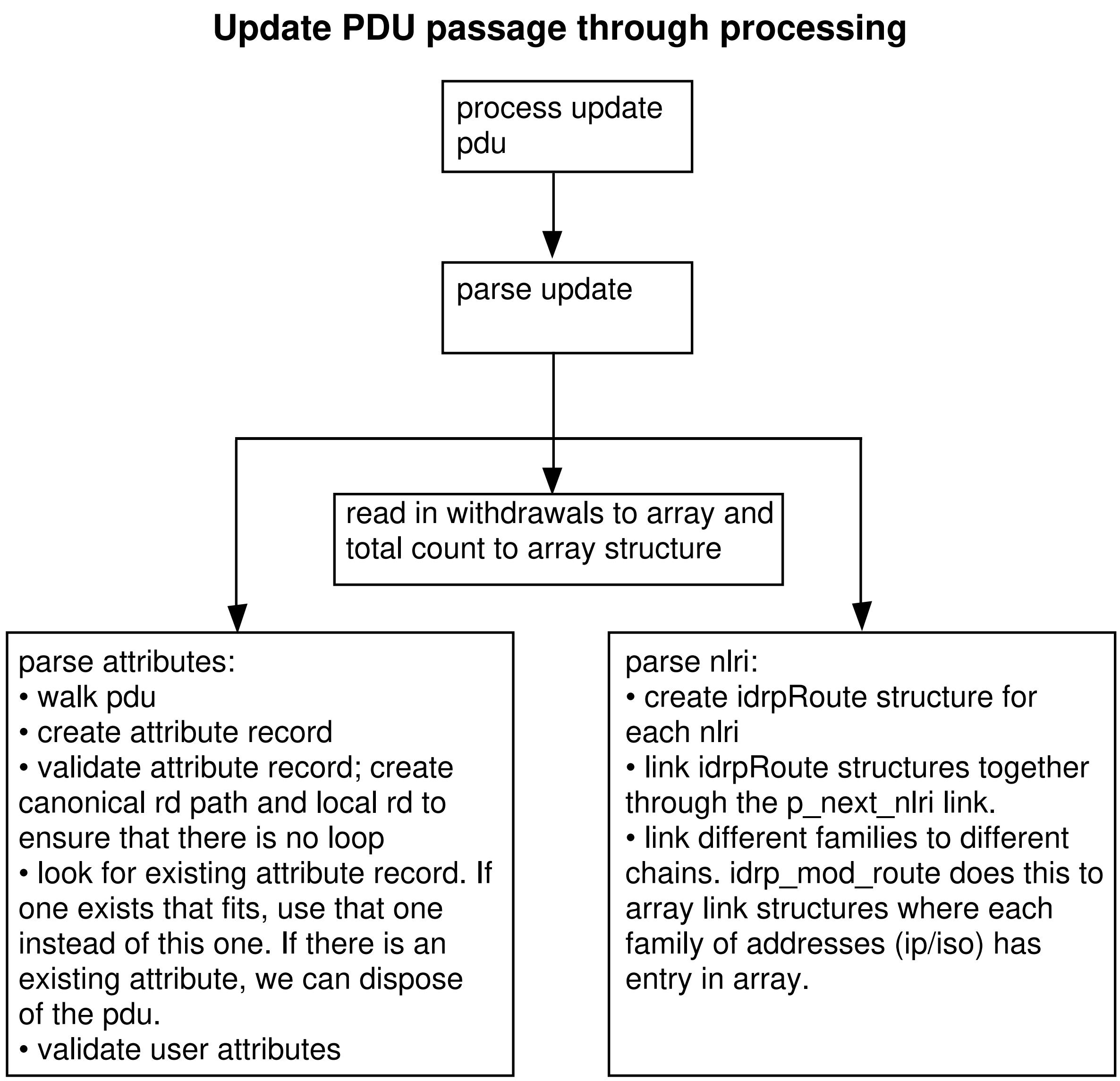


Figure 13 — Update PDU parsing

routine: parse_update_pdu

Actions:

1) Withdraw IDs are parsed

The Withdraw route IDs are parsed out of the UPDATE BISPDU into an withdraw array structure for further processing.

2) Attributes are parsed

5) parse_update_cleanup(results): if valid, then idrp_process_pdu results

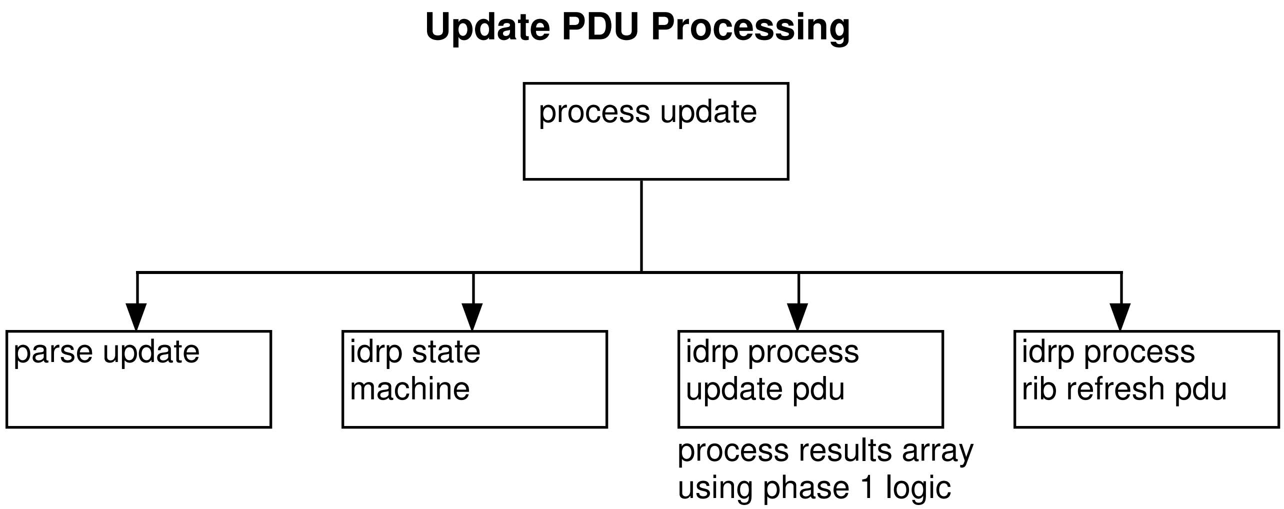


Figure 12 — Update PDU processing

4. Program flow description for update PDUs

The UPDATE BISPDU contains the routing information sent from BIS to BIS. The BISPDU is parsed the routes it carries entered into the gated routing table. The routes are then propagated to BIS neighbors that need to know about routing changes. The IDRP specification (ISO 10747) describes the processing of routing information in three phases. Below we describe how the UPDATE PDU is parsed and the three IDRP processing phases are done.

4.1. Overview of IDRP UPDATE parsing code

The process update function in the IDRP code (process_update) calls the update parsing function (parse_update). The parsing routine walks through the UPDATE BISPDU finding the withdrawal route IDs, attributes and network layer reachability information (NRLI) in the PDU. The parsing routine returns to the process_update function an indication whether the BISPDU was successfully parsed or encountered a problem while being parsed, plus a structure containing results of the parse.

The process_update function calls the IDRP state machine with either the IDRP_UPDATE or IDRP_UPDATE_ERROR event. If the update is received in a valid state and should be processed, the process update code checks to see if it is in the middle of a RIB-Refresh sequence. If this PDU is one of a set of update PDUs for a RIB-Refresh, a refresh information structure is created to hold the parsing results from the UPDATE PDU and a pointer to the UPDATE PDU. This refresh information structure is linked to a refresh processing list. When the refresh is completed, each UPDATE PDU is processed via the idrp_process_pdu_routes.

If the update is received outside of a RIB Refresh sequence, idrp_process_pdu_routes is called immediately. Figure 12 shows the logic of the update.

4.2. Process update routine descriptions

4.2.1. process_update_pdu

routine: process_update_pdu

calling parameters: peer, PDU, length

peer - idrpPeer structure of peer sending this PDU

PDU - pointer to PDU

length - length of PDU

Actions:

1) call parse PDU

2) if PDU valid, then calls state machine.

3) if refresh cycle, links results of parse to refresh list for later processing

4) if invalid state, releases results from parse

buffer - pointer to output buffer that contains the BISPDU to be transmitted

Actions:

1) fill in flow control fields
2) Generate the authentication checksum
3) copy buffer to task_send_buffer
4) use gated task_send_packet to send PDU
5) put buffer on retransmission queue if sequenced

3.3.3. post_enqueued_pdus

routine: post_enqueued_pdus

calling parameter: peer

peer - idrpPeer Structure to see if we can send queued BISPDUs.

Action:

Loop trying to send all PDUs on the transmit list. Each PDU is sent by calling the idrp_post routine. If the peer becomes flow blocked, stop.

BISPDU transmission

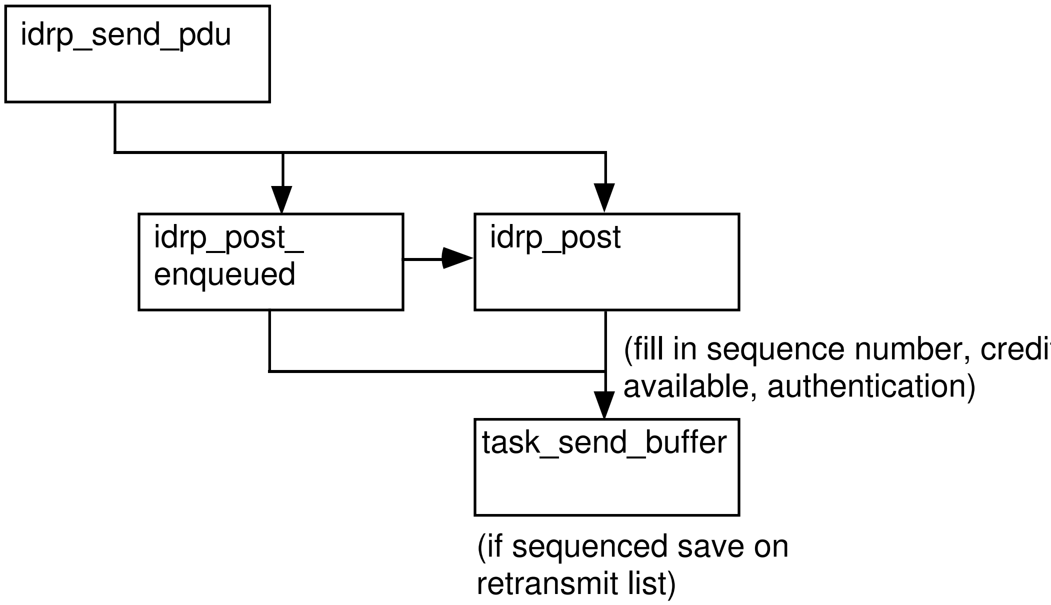


Figure 11 — BISPDU transmission

3.3.1. idrp_send_pdu routine

routine: idrp_send_pdu

calling parameters: peer, PDU, type, length

peer - idrpPeer structure of BIS neighbor to which we are sending the PDU

PDU - pointer to the idrpPdu structure containing the BISPDU

type - type of BISPDU (e.g. OPEN)

length - length of the PDU

Actions:

1) fill in the BISPDU header with PDU, length type, source address and destination address
2) allocate Output buffer structure and fill it in
3) Determine if buffer is sequenced. If sequenced, link PDU to transmit list and call post_enqueued_pdus. If not sequenced, call idrp_post routine

3.3.2. idrp_post routine

routine: idrp_post

calling parameters: peer, buffer

peer - pointer to idrpPeer structure of BIS neighbor to which we are sending the BISPDU.

unreachable - Flag to indicate whether the only thing to be sent is a list of route IDs that are now unfeasible.

Action:

1) get memory block to put UPDATE PDU in
2) fill in BISPDU with unreachable Route IDs
3) if unreachable flag is set, just call idrp_send_pdu
4) if unreachable flag is not set,
   4.1) fill in attributes from attribute record
   4.2) fill in NLRI section one protocol at a time.
5) call idrp_send_pdu to send the UPDATE PDU

3.2.7. Sending echo PDU

routine: send_echo_pdu

calling parameter: peer

peer - The idrpPeer structure of the BIS neighbor to which we are sending the optional echo BISPDU.

Action:

Allocate task memory to send ECHO PDU. Call idrp_send_pdu to send the PDU.

3.3. Transmitting BISPDUs

The transmission of any BISPDU from IDRP code goes through the idrp_send_pdu routine. The idrp_send_pdu routine adds most of the fixed header. If the PDU requires IDRP transport sequencing, the BISPDU is queued for sequence transmission via the post_enqueued_pdu routine. This routine checks the transmission queue for pending PDUs and sends them if the connection is not flow blocked. When a BISPDU is to be sent from the queue the idrp_post routine is called. The idrp_post routine fills in the sequence number, credit authorization, and does the authentication checksum. The idrp_post routine then calls the gated functions to send a buffer on a socket. The task_send_packet routine is used to send the buffer. The figure below illustrates this flow:

peer - The idrpPeer structure of the BIS neighbor to which we are sending the keepalive.

Action:

Create KEEPALIVE PDU. If the remote side is not flow blocked, send the PDU via the idrp_send_pdu routine.

3.2.4. Sending ERROR PDU

routine: send_error_pdu

calling parameters: peer, code

peer - The idrpPeer structure of the BIS neighbor to which we are sending the ERROR PDU.

code - The error code for the ERROR PDU to be sent to the BIS neighbor. This error code is used for double-checking that the error information found in the idrpPeer structure in the last_error_pdu is correct.

Action:

Fill in the ERROR PDU using the last_error_pdu information found in the idrpPeer structure. After body of the ERROR PDU is built, call idrp_send_pdu to send it.

3.2.5. Send CEASE BISPDU

routine: send_cease

calling parameter: peer

Action: Allocate memory for CEASE and call idrp_send_pdu.

3.2.6. Send UPDATE PDU

routine: send_update

calling parameters: peer, p_send_list, unreachable

peer - pointer to idrpPeer structure

p_send_list - IDRP send list which contains:

rib_id - identifier for the RIB (always zero for now)

p_next - IDRP send list link (not used by routine)

ann_nlri[] - array of pointers to linked list of idrpRoute structures. Each idrpRoute structure has one NLRI for a particular family. The ann_nlri array has a list per family type such as ISO or IP.

p_attr - pointer to attribute record

withdrawal structure - an array of route IDs to be withdrawn plus a count of the number of route IDs.

3.2.1. Memory allocated for outbound PDUs

The outbound PDUs are allocated out of gated task memory. The outbound PDUs are linked together on the transmit and retransmission queue by an IDRP output buffer structure. This output buffer structure has:

1) a link for the transmit or re-transmit queue
2) a pointer to the PDU structure
3) an indication of the PDU's length
3) an indication of whether the PDU is sequenced or not
4) a reference count

Figure 30 illustrates this structure.

3.2.2. Sending OPEN PDU

routine: send_open_pdu

calling parameter: peer

peer - pointer to idrpPeer structure of the BIS neighbor to which we are sending an open.

Actions:

The peer parameter points to a structure containing the information we need to build the OPEN PDU. The IDRP specification indicates what these parameters are. Once the PDU is built, the send_idrp_pdu routine is called.

An optional open sequence outside the IDRP is available for the user. In this optional open sequence, the router will send multiple OPEN PDUs prior to timing out with the HOLDTIMER and returning to close state. This rapid transmission of opens may help a connection come up more quickly if one side executing the normal open sequence and the second side is using this fast open sequence.

The open sequence is controlled by the max_open_sent value in the idrpPeer structure. If this value is non-zero, the implementation will set the open_sent timer after sending the OPEN PDU. If this timer expires, the OPEN PDU will be retransmitted max_open_sent times prior to executing the transition to the CLOSE state from the OPEN_SENT state. Both the max_open_sent and the time between OPENs are set in the configuration file. These timers may be set either per neighbor or for a group of neighbors.

The only RIB attribute supported at this time is the default RIB. Also, no Routing Domain Confederation (RDC) support is available.

These functions will not change during the first two releases of IDRP. However, this portion of the document will change when RDCs or multiple RIBs are supported.

3.2.3. Sending KEEPALIVE PDU

routine: send_keepalive_pdu

calling parameter: peer

RIB Refresh Structures

idrpRefresh structure
pointer to next Refresh structure on list (Refresh list pointers are kept in the idrpPeer structure in “refresh” pointer.)
pointer to update pdu is included in refresh
parse results array for the pdu processing of this update pdu

Refresh_info structure
sequence number of RIB refresh start PDU
head of idrpRefresh PDU structure
tail of idrpRefresh pdu structure
sequence number of RIB refresh end PDU
count of PDUs to process
RIB ID

Figure 9 — Rib Refresh structures awaiting processing

3.2. Outbound BISPDU processing

Outbound BISPDUs are sent as a result of a transition in the IDRP state machine, or a timer expiring, or phase 1 processing of the UPDATE PDUs or phase 3 processing of the routing table changes. Figure 10 shows the links between the IDRP state machine, timers, phase 1 processing and phase 3 processing and the routines used to send the PDUs.

Outbound BISPDU processing

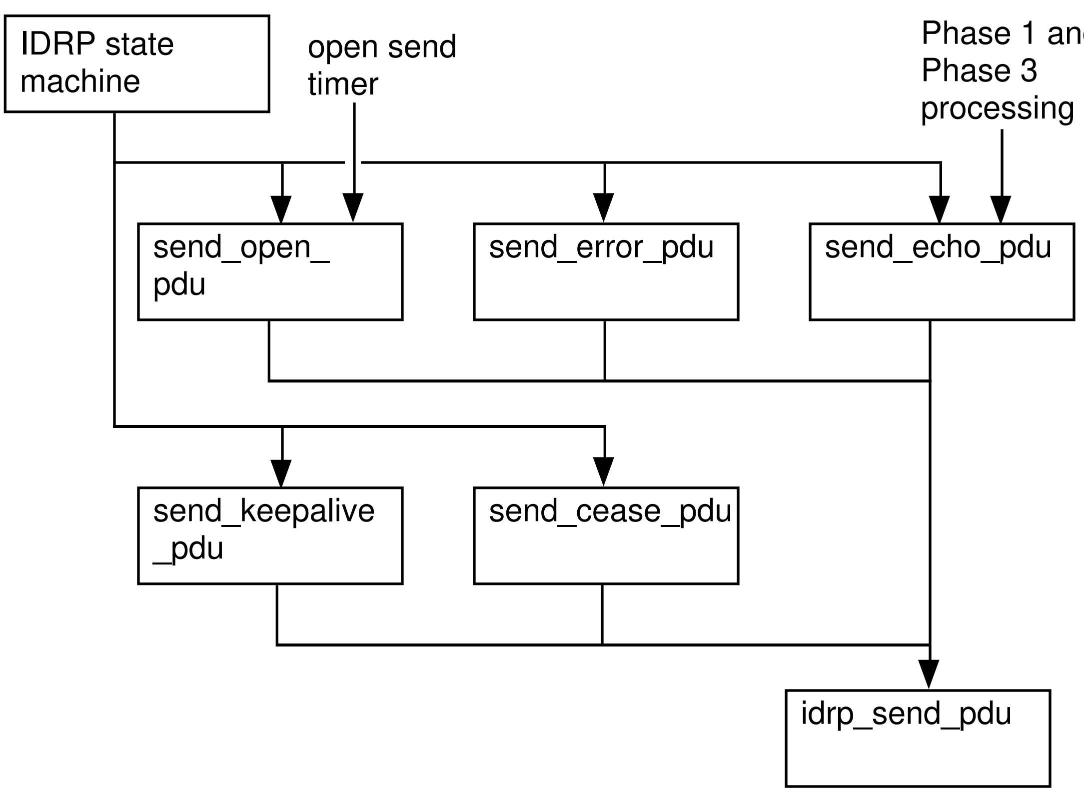


Figure 10 — Outbound BISPDU processing

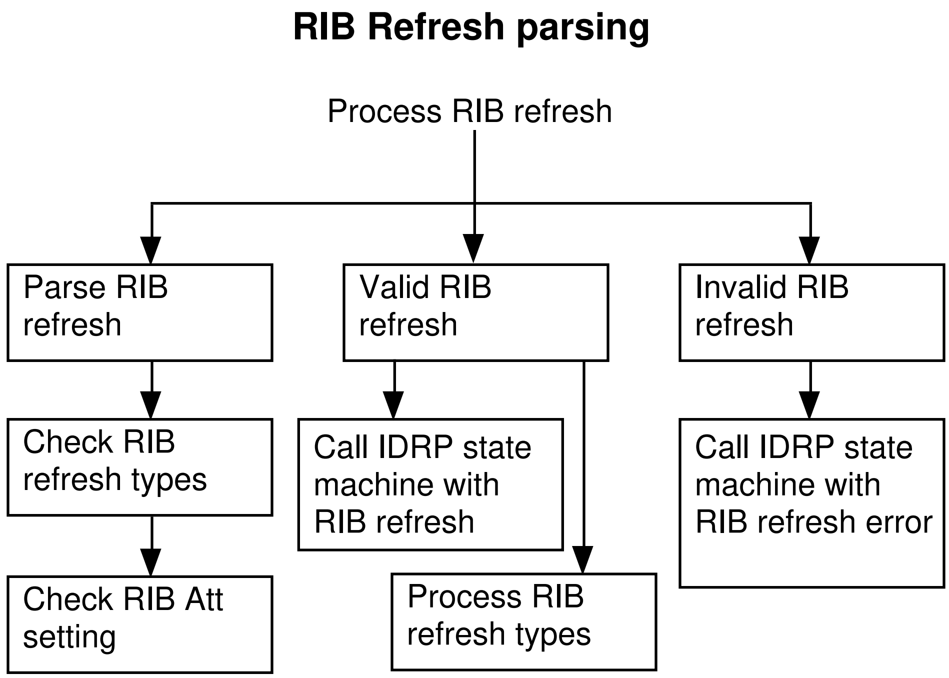


Figure 7 — Rib Refresh parsing

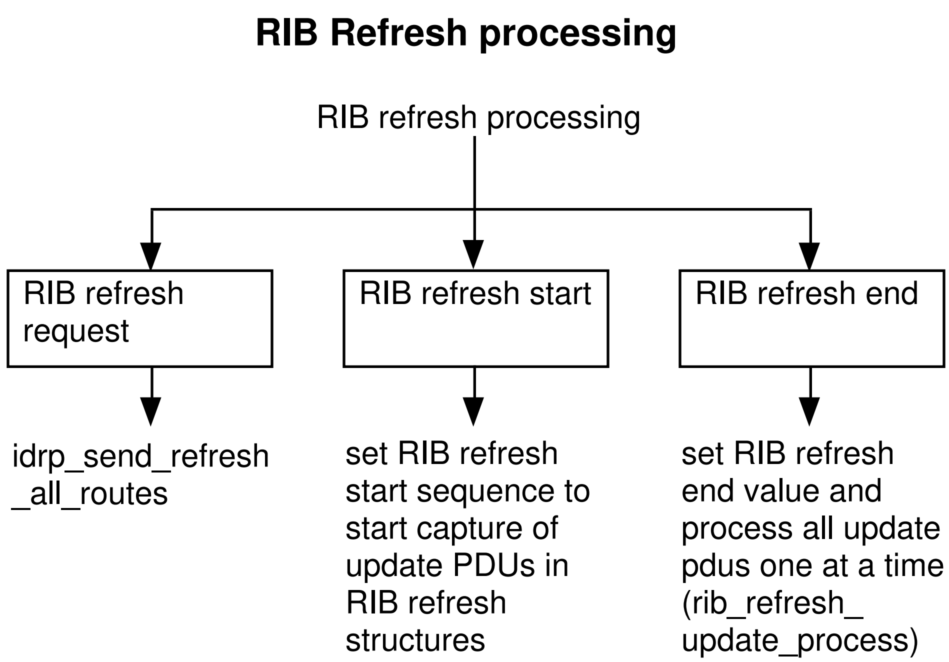


Figure 8 — Rib Refresh processing

Error BISPDU parsing and processing

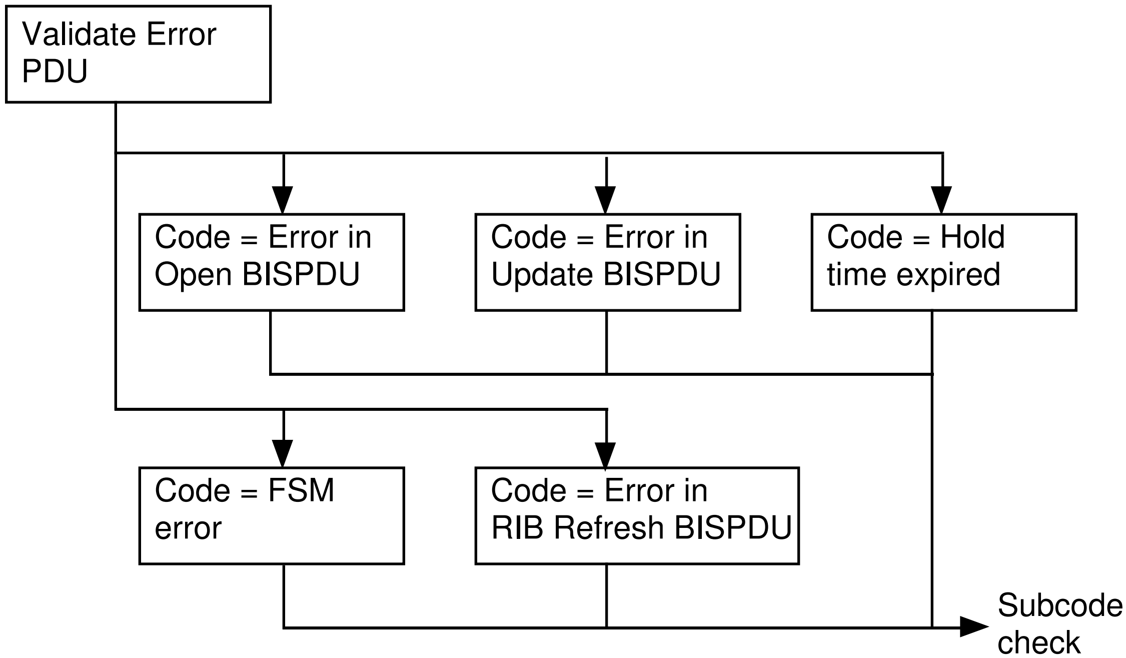


Figure 4 — ERROR BISPDU parsing and processing

Cease BISPDU parsing and processing

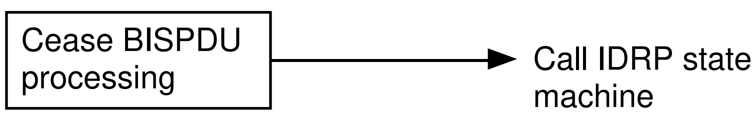


Figure 5 — CEASE BISPDU parsing and processing

Keepalive BISPDU parsing and processing

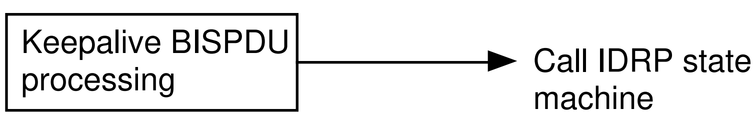


Figure 6 — Keepalive BISPDU parsing and processing

OPEN BISPDU processing

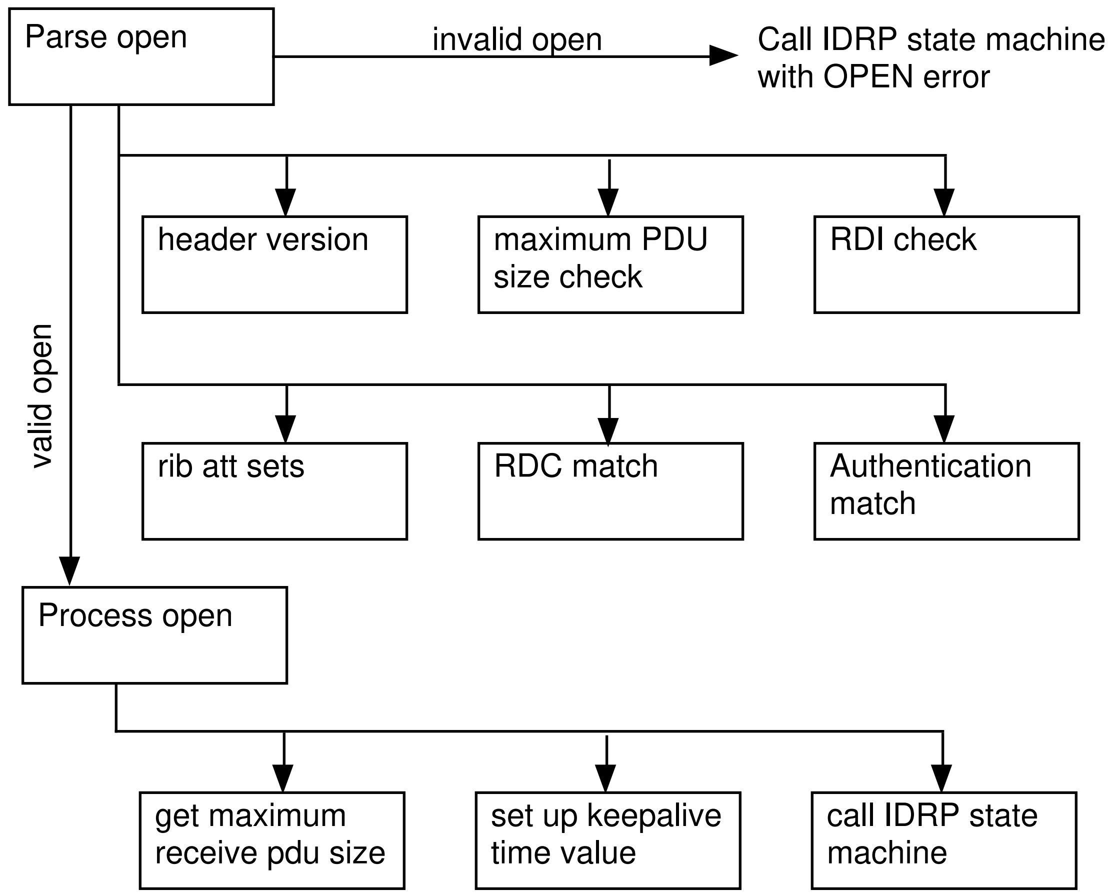


Figure 3 — OPEN BISPDU parsing and processing

idrp_recv_pdu processing

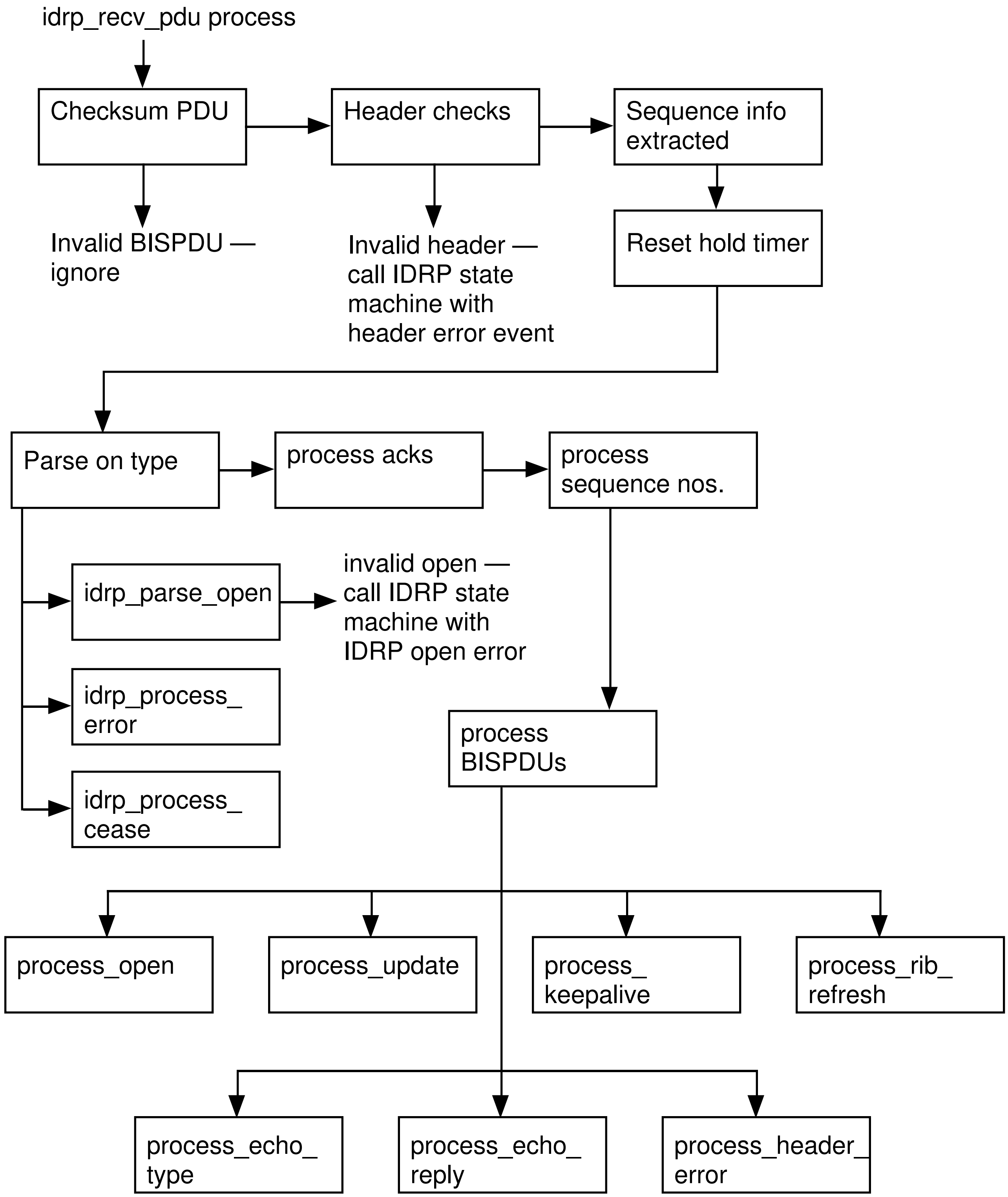


Figure 2 — idrp_recv_pdu processing

IDRP packets' passage through IDRP code

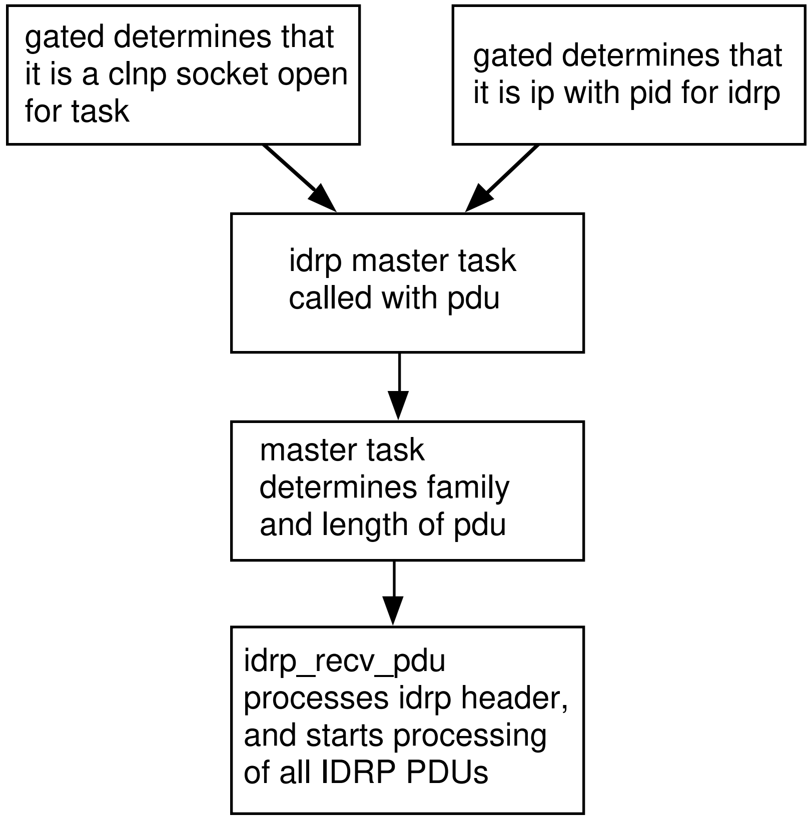


Figure 1 — IDRP packets' passage through IDRP code

3. Program control flow description for all non-update BISPDUs

3.1. Inbound BISPDU processing

Below is a generic description of how the PDUs are passed from gated to the IDRP master task that receives the PDU and on to individual BISPDU processing. The descriptions of the processes are done in diagrams.

5) IDRP hash table for inbound routes

6) IDRP parsing structures

The IDRP withdraw and announce list structures are allocated out of task memory.

7) IDRP refresh structures

The IDRP refresh structures store the results of an UPDATE PDU parsing for later processing at the end of a RIB-Refresh update. They are allocated out of task memory.

2.5. Reconfiguration of policy

The gated code provides task_cleanup, task_reinit and task_newpolicy links to the IDRP task for both the master task and the peer task. In the second phase of the Merit implementation, reconfiguration of policy will be supported.

(Section 2.5 - will be further developed during the 2nd phase of the Merit implementation.)

2.4.1. Configuration memory allocation

The IDRP code requires configuration information about IDRP BIS neighbors (also called peers), the local BIS configuration, and local IDRP routes. This information is read out of the gated configuration time before the IDRP tasks are created. Memory must be allocated to store this information.

IDRP allocates memory to store this configuration information by calling one of the following routines depending on the size of the block needed:

• idrp_peer_alloc - IDRP Peer information (idrpPeer structures)
• idrp_med_blk_alloc - Local Route structures (idrpRoute structures for local routes, and idrp_attribute_record structures for local routes' IDRP attributes)
• idrp_small_blk_alloc - Other Local route structures (IDRP Routing Domain pathway structure (idrp_canon_rdpath), and IDRP route list structure (idrpRoute_entry))

In turn these routines call two gated routines for allocation of memory outside of a task: task_block_alloc and task_block_init. The task_block_init routine specifies a size of structure the gated routines will allocate. The task_block_alloc allocates memory of a size specified by a previous task_block_init call.

In addition, in the case of re-configuration some IDRP peer or IDRP local routes may disappear. The following idrp_routines free the configuration memory: idrp_big_blk_free, idrp_med_blk_free and idrp_small_blk_free. In turn these routines call the gated routine task_block_free.

2.4.2. IDRP task block memory allocation

The IDRP code uses the task memory allocation routines both to allocate memory for structures needed in parsing BISPDUs into IDRP route structures and to allocate memory for structures needed in sending BISPDUs. The structures allocated from task memory include:

1) The IDRP route structure

The IDRP Route structure (struct idrpRoute) holds one destination found in the network layer reachability information (NRLI) received from one peer.

2) IDRP attribute record

The IDRP attribute record structure contains a unique instance of a set of IDRP attributes. An attribute record may have many idrpRoute structures linked to it from one or more BISPDUs.

3) Ordered list of RDIs

IDRP RD path information is a sequence or set of Routing Domain Identifiers. These Routing Domain Identifiers are kept in an ordered list of RDIs to help in checking for duplicate RDs and to allow easy handling for policy routines.

4) IDRP output buffers

IDRP buffers are allocated to store BISPDU information prior to transmitting a BISPDU to a neighbor.

- USR2 signal
- INT signal
- CHLD signal

Incoming PDUs do not interrupt gated. The gated code (in task.c) simply does a “select” on the socket and waits for one or more sockets to be ready to read or write.

Gated handles the kill signal by calling routines in each task to terminate the protocol functions. The routines are called the task_terminate functions. In IDRP, there is a master task for receiving packets from the operating system and a task for each peer to handle the timers required for each connection. The terminate routine for the master task is idrp_terminate. The terminate routine for an IDRP peer is idrp_peer_terminate.

The alarm interrupts provide gated with its timer service. A discussion of the timers used by IDRP is found in the next section.

The gated routines that handle the USR1 interrupt allow the user to change the information that is logged by gated. For example, if additional tracing for a peer is required, the USR1 interrupt could be sent to gated, requesting it to change the trace flags on the IDRP protocol for that peer.

The gated routines that handle the USR2 interrupt routines check for new interfaces. If a new interface has been added, allowing the IDRP protocol to send BISPDUs to a new neighbor BIS, the USR2 will request gated to enable the interface, in turn allowing the IDRP protocol to open a BIS-BIS connection through that interface.

2.3. Gated timer functions usage by IDRP

IDRP creates the following timers for each peer task: keepalive, closewait, hold, retransmit, start, open sent, minimum route advertisement, and an optional debug echo timer. The keepalive, closewait, hold, retransmit, and minimum route advertisement timers are specified in the IDRP protocol specification.

The start timer provides a mechanism to queue the start event after a peer connection has been configured and initialized. After the start timer expires, the start event is executed in the IDRP state machine.

The open sent timer allows for multiple opens to be sent in rapid sequence prior to holding down the connection until the next open sequence.

An optional echo function requires a timer. This echo function allows two BISes to exchange an ECHO PDU to test the connection at a BISPDU level. It is useful for debugging.

IDRP uses the gated functions timer_create, timer_set and timer_reset to create, set, and reset timers.

2.4. Gated memory allocation used by IDRP

The IDRP code uses two types of memory allocation from the gated structures: task memory allocation and allocation of memory during the configuration cycle of gated.

2. Data flow description

Gated provides a single-threaded event driven environment for implementing routing protocols. Events are generated by sockets being ready for read, write or exceptions, interval timer expiration and signals requesting re-configuration and shutdown. Threads are non-interruptable. The gated code provides support for socket handling, interrupt signals from the UNIX system, timers, memory management and re-configuration of routing policy.

Gated interacts with the IDRP code (and all protocol code) by means of a number of entry points the protocol must register with gated. Of particular interest is the "flash" routine, in this case idrp_flash. Gated tracks the routes a protocol has expressed interest in (in IDRP's case, the routes IDRP has installed and/or propagated) and calls the protocol's flash routine to alert the protocol when any such route changes; that is, is added or deleted. Likewise, when a protocol has made a change to the routing table it wishes to inform gated of, it must flash gated.

At a very high level, the path an IDRP PDU takes through gated looks like the following:

• PDU arrives
• IDRP processes PDU
• IDRP flashes gated
• Gated modifies its routing table (i.e. the IDRP Loc_RIB)
• If the new route is the preferred route, gated flashes IDRP
• IDRP propagates the reachability information to its peers

2.1. Gated socket handling for IDRP

The IDRP idrp_init routine opens a master task to receive BISPDUs from the appropriate socket. The IDRP code using CLNP will use a specially defined IDRP socket which checks the first byte of the transport payload to see if the IDRP protocol is needed. For a more complete description of the IDRP socket, please see Appendix A. A separate task is created for each IDRP peer to handle timers specific to each BIS-to-BIS connection.

The IDRP master task, idrp_recv, handles reading data from the socket by calling the task_receive_packet gated routine. The idrp_recv code verifies that the PDU is an IDRP BISPDU from a configured neighbor. If the packet is valid it is passed to the idrp_recv_pdu routine for processing.

The IDRP idrp.c code sends a PDU packet by calling the "task_send_buffer" gated routine with the BISPDU contained in the buffer.

2.2. Gated interrupt functions used by IDRP

Gated is driven in two ways - signals and polling. Signals are used for reconfiguring gated routing policy or interface configurations or to change what logging is used. The PDUs received by gated are received by a select in task_main that blocks until a socket is ready for reading or writing.

The gated code catches the following interrupts from UNIX systems:

• UNIX kill signals,
• alarms (used for gated timers),
• USR1 signal

idrp_sock.c	all routines to send and receive over IP/ISO	idrp_ip_raw_sock_recv idrp_clnp_sock_recv idrp_idrp_sock_recv idrp_clnp_send_pdu idrp_send_idrp_pdu
idrp_timers.c	timer routines: keepalive, holdtime, start timer (implementation specific), re-transmit timer	start_rexmit_timer kill_rexmit_timer start_keepalive_timer start_opensent_timer
idrp_transport.c	IDRP End to End Transport code	flush_xmit path_down free_acked_pdu flush_rexmit_queue close_peer begin_close ack_pdu
idrp_validate.c	Validation routines for IDRP attributes	checksum_ok idrp_rib_id

The generic ISO routines will be moved into common files with other routines which support the ISO functions in gated. Currently, the routines in IS-IS and IDRP are separate due to these protocols having been newly added to gated. Future versions of the IDRP Design Specification will reflect changes as they are made to the code.

idrp_rt_phase3.c	Phase 3 processing of IDRP routes	idrp_flash phase3 idrp_phase3_dump phase3_status_change ph3_status_case1 ph3_status_case2 ph3_status_case3 ph3_status_case4 ph3_status_case5 ph3_status_case6 ph3_status_case7 ph3_status_case8 idrp_send_phase3_routes phase3_newpolicy
idrp_rt_phase_util.c	Utility routines for Phase 1 and Phase 3 processing	link_ann_list link_send_list send_with_attr send_update_reset_send send_nlri_attr create_send_list flush_att_send_list idrp_send_list_room idrp_del_sent_routes best_ext_routes idrp_add_route_locate idrp_with_route_locate idrp_rt_change free_rt_chain_walk
idrp_rt_policy.c	Policy routines	PREF DIST DIST_LIST_INCL DIST_LIST_EXCL DIST_ATTR idrp_gated_import idrp_gated_export (most of these routines will be shells until 2nd stages of IDRP code)
idrp_rt_util.c	General utilities in code for hash table, idrpRoute allocation and de-allocation, idrpRoute_entry structure allocation and deallocation	idrp_rid_hash idrp_add_hash_entry idrp_free_hash_entry idrp_release_hash_tbl idrp_init_hash_tbl idrp_alloc_idrpRoute idrp_free_idrpRoute idrp_free_idrpRoute_entry idrp_add_idrpRoute_entry
idrp_sm.c	IDRP state machine code	idrp_sm

idrp_rt_minadv.c	Processing with Minimum Route Advertisement Timers: • for other domains • for this RD	Other RD routes: idrp_add_min_route_adv advmin_interval_calc idrp_set_min_route_timer idrp_process_minadv idrp_min_adv_rt Within this RD routes: idrp_add_rt_min_advRd idrp_set_min_advRD_time idrp_process_minAdvRD idrp_min_advRD_rt
idrp_rt_peer.c	Processing by IDRP peer: peer up, peer down, consistency check on AdjRib, validation of AdjRib, peer reinitialization after configuration change, send refresh of AdjRib to peer	consistency_check validate_AdjRib validate_allRibs idrp_rt_send_init idrp_send_refresh_all_routes idrp_refresh_allpeers peer_down peer_up idrp_peer_route_pull idrp_refresh_all_peers
idrp_rt_phase1.c	Phase1 processing of IDRP routes	Preferences: idrp_pref idrp_ext_info_pref PDU processing: idrp_route_mod idrp_process_pdu_routes IDRP phase1 processing: idrp_phase1_processing phase1_internal phase1_external ph1_with_int_route ph1_with_rep_int_route ph1_add_int_route ph1_with_ext_route ph1_with_rep_ext_route ph1_add_ext_route send_best_ext del_route_best_ext remove_ann_dup
idrp_rt_phase2.c	Phase 2 processing of IDRP routes, Phase 2 tie breaking code, IDRP to gated preferences	tie_break tie_break_iso iso_next_hop_compare iso_multi_exit_compare

idrp_pdus.c	routines that send PDUs	idrp_post post_enqueued_pdus idrp_send_pdu send_open send_update_pdu idrp_send_error send_cease send_keepalive
idrp_rib_refresh.c	routines to handle IDRP RIB REFRESH PDUS and processing	rib_refresh_pdu_release refresh_pdu_link rib_refresh_update_process
idrp_rt.c	IDRP routing table general functions (reinit style code)	sock_supported idrp_sock_dup
idrp_rt_ext_info.c	routines to handle transfer of external information to IDRP Includes: IS-IS <-> IDRP static <-> IDRP interface <-> IDRP BGP <-> IDRP	idrp_find_ext_info_rt find_ext_info_attr_rec create_ext_info_attr_rec idrp_aspath_to_canon_rdpath generic_ext_info idrp_as_rdi_map idrp_bgp_within_RD idrp_ext_info_peer
idrp_rt_iso.c	generic ISO processing routines (hopefully most of these will be subsumed into gated routines.)	idrp_iso_sockaddr_mask idrp_set_iso_sockun sockun_to_prefix sockun_toa
idrp_rt_local.c	Processing of local IDRP routes configured into gated	idrp_add_local_rt gated idrp_local_rt find_local_route dest create_local_attr_record create_local_ext_info_att_rec idrp_options_to_mask link_local_attr idrp_fill_local_man_opt_attrib fill_next_hop fill_DIST_LIST link_local_attr_list create_local_rd_path idrp_create_local_route_reset idrp_flag_local_att idrp_local_route_clean idrp_clean_local_att idrp_free_local_route_chain idrp_free_local_rt idrp_local_mem_fit

md4.h	MD4 data structure definitions	Data structures used in MD4 generation.
osi.h	ISO Address structure definitions not yet in iso.h	ISO prefix (bit count plus ISO address), ISO address (byte count plus ISO address)
idrp.c	general trace routines and debugging routines	drop_packet idrp_trace log_nm_event free_buffer proto_to_family family_to_nrli_id idrp_send_dest_set
idrp_att_rec.c	routines dealing with the IDRP attribute record	valid_attr valid_usr_attr add_local_rd_to_path valid_seg_type valid_rd_path link_rd_canon find_attr_rec compare_atts idrp_dif_path ATTS_REC_ZERO idrp_find_routeid_chain idrp_free_nlri_att_rec idrp_free_attr_rec
idrp_attrib.c	routines parsing and validating IDRP attributes	
idrp_dump.c	Routines that dump IDRP structures under gated trace or logging signals	idrp_master_dump idrp_peer_dump idrp_peer_rtbit_dump dump_local_rib dump_rib(peer)
idrp_init.c	initialization code for IDRP module: gated tasks, reinitialization of connections, idrp_newpolicy	idrp_init, idrp_reinit_peer idrp_cleanup idrp_newpolicy
idrp_parse_pdu.c	routines that parse BISPDU for IDRP code	idrp_parse_open process_open parse_update parse_update_cleanup process_keepalive process_error process_cease parse_rib_refresh process_rib_refresh process_echo

idrp_init.c	IDRP initialization routines	IDRP global variables Gated based initialization routines: idrp_cleanup idrp_var_init idrp_init idrp_newpolicy IDRP peer structure handling routines: idrp_peer_alloc, idrp_med_blk_alloc, idrp_small_blk_alloc, idrp_delete, idrp_terminate, idrp_peer_cleanup, idrp_peer_reinit, Generic ISO routines: pretty print routines for dumps or log file: iso_ptoa, iso_ntoa, Comparison routines: compare_iso_addr, isopfxcompare,
idrp_rt.c	IDRP routing table routines	IDRP hash table routines IDRP Route structure handling routines IDRP Route list handling routines Validation of AdjRib Consistency check of gated routing table IDRP decision routines for Phase 1 processing Phase 2 processing Phase 3 processing Routines to add routes to gated tables Routines to send initial blast of routes Routines to handle local routes
md4.c	IDRP validation code (MD4 message digest code)	Encode, decode routines for MD4, MD4 initialization, MD4 update
md4global.h	Global Type definitions used by MD4 routine	Global type definitions

1. Program structure

Merit's IDRP implementation uses a modular structure similar to that used by other protocols in gated: an initialization module, a PDU processing module, and a routing table module. The IDRP code is found in the following modules:

NOTE: This section is slightly out of date. I have included the file list below. I will update it shortly.

Module Name	Function	Routines included in module
idrp_proto.h	IDRP protocol definitions	IDRP structures to decode BISPDUs and values used in parsing the BISPDUs
idrp_globals.h idrp_globals.c	IDRP global variable definitions	-
idrp_macros.h	Macros for IDRP code	Macros try to improve readability of code
idrp_prototypes.h	prototypes for all routes	prototypes are good place to look at routines
idrp.h	IDRP data structure definitions	Structure for each IDRP route Structure for each IDRP peer Structure to store attributes found in UPDATE PDUs Structures used to pass information found in BISPDUs to routing table processing (route announce list or Withdraw route structure) Prototypes of all IDRP functions
idrp.c	IDRP protocol and timer PDUs	Timer routines BISPDU sending routines IDRP transport handling Peer up and peer down routines Parsing of incoming BISPDUs IDRP events IDRP state machine PDU checksum and header validation routines Attribute handling routines

changes to the gated table. IDRP phase 1 processing and portions of the Phase 2 processing are done on the a per BISPDU basis. However, the rest of phase 2 and phase 3 may run over many BISPDUs. Because many people using this IDRP protocol may be porting the Merit code to other environments, alternative design choices are provided to allow others to select what portions of the IDRP code they would like to use.

This design document contains descriptions of the:

- Internal structure of the IDRP code (Section 1)
- Data flow of BISPDUs between gated and the IDRP code (Section 2)
- Program control flow description for all BISPDUs, except for the processing of the UPDATE BISPDU (Section 3)
- Program control flow description for processing the UPDATE BISPDU and adding routes to the routing table (Section 4)
- Descriptions of data structures and algorithms (Section 7)
- Alternative designs (Section 10)

Introduction

The Merit IDRP implementation in gated has been influenced heavily by the internal structure of gated. An implementation of the IDRP protocol could be broken into three parts:

- IDRP BISPDU processing
- Routing table updates and processing
- 8473 forwarding

In the Merit implementation of IDRP, the IDRP BISPDU processing has been made as independent as possible. However, the routing table code is closely tied to the gated routing structure. The 8473 forwarding engine depends on the insertion of routes by gated into the underlying operating system.

The gated routing structure is designed to allow multiple protocols to share routing information. Gated's routing tables retain all information for a given destination (such as a CLNP network) received from any protocols the gated program supports. A gated routing table may contain IS-IS routes for a destination, IDRP routes for a destination, or both. Gated provides the mechanism to decide which protocols' routes will be installed in the forwarding engine for this router.

The IDRP Adjacency RIBs are contained in the gated tables and not in separate structures. The gated routing tables are linked both by destination and by the gateway that sent the route. The Adjacency RIBs are found by searching the gated tables for any routes linked to a particular peer for the IDRP protocol. In gated the "LOC_RIB" (active routes in gated terms) contains routes from not only the IDRP protocol, but any other protocols.

The gated routing code also provides a mechanism for passing the routing information between the two protocols via the gated routing table. The gated routing table is structured to allow many protocols to share routing information.

Gated's routing table is not the only structure that could allow for sharing of routing information between IS-IS and IDRP. Alternate structures were examined for this project, but the gated structure chosen for consistency within the gated framework. It is hoped that this consistency will provide a good prototype platform and allow for exchange of routing traffic in the Internet environment, where multiple types of network reachability information, such as IP and CLNP, must be supported. The current technical experts working on gated in the Internet encouraged this structure because of the dual stack nature of the Internet. However, in an OSI only environment other structures for the routing table may be more efficient.

Because of our understanding that the BISPDU handling portion of this code may need to migrate to a non-gated environment, the data structures chosen for the IDRP routes contain some duplication of information found in the Gated tables. This duplication may allow the code to be removed from the gated structure and integrated into a different routing table framework.

Gated runs in a single threaded event driven environment which processes a task until it is done. Events are generated by sockets being ready for read, write or exceptions, interval timer expirations and signals requesting re-configuration or shutdown. Threads are non-interruptable and therefore care must be taken to avoid excessive processing time in a thread (task) when possible.

Exceptions to this mode are rare, and tightly controlled. For example, the single threaded environment encourages processing of a BISPDU from start to finish instead of bunching

Figures

Table of Contents

Brief Table of Contents

Design Document for Merit IDRP Implementation

version 2.3

6/30/93

Susan Hares
John Scudder