Nimrod Functionality and Protocol Specifications (Draft)

Contents

1 Introduction

Nimrod is a scalable routing architecture designed to support a dynamic internetwork of arbitrary size, to provide service-specific routing in the presence of multiple constraints, and to admit incremental deployment throughout an internetwork. The key features of Nimrod include representation of internetwork connectivity and services in the form of maps at multiple levels of abstraction; source- and destination-controlled route generation and selection based on maps and traffic service requirements; and source- and destination-controlled message forwarding according to the routes selected.

At the most general level, one may view any routing system as a set of basic functions which are producers and consumers of certain databases of routing information. These routing functions and their associated routing information include:

1. Assembling, distributing, and collecting information necessary for route generation and selection. This information includes internetwork connectivity and services, traffic service requirements, and locations of traffic sources and destinations.

2. Generating and selecting routes, based on the collected information.

3. Establishing in routers information necessary for forwarding messages, based on the selected routes.

4. Forwarding messages along these routes.

Routing systems may, however, differ in the details of the mechanisms that provide a particular routing function. As Nimrod has been designed for routing in large, heterogeneous, and dynamic internetworks, its basic routing functions include additional mechanisms for reducing the quantity of routing information that must be distributed, processed, and stored throughout an internetwork; for discovering and accommodating changes in routing information caused by physical changes in an internetwork; and for protecting the integrity of routing information.

This document contains a specification of some of the functionality and protocols comprising Nimrod and is organized as follows. In the remainder of this section, we give an overview of the Nimrod Architecture, entities, and agents. In section 2, we give an overview of the key Nimrod operations. In Section 3 we describe the databases that are manipulated by Nimrod. Following that is a description of each of the Nimrod protocols - Reliable Transaction protocol (section 4), Path Management protocol (section 5), Hierarchical Update protocol (section 6), Hierarchical Query-Response protocol (section 7), and the Discovery protocol (section 2.6). The document also contains an appendix that contains figures (only if this is the .ps version) and other items that are useful but not crucial to the understanding of Nimrod functionality and protocols.

1.1 Overview of the Nimrod Architecture

Before Nimrod routing can be applied within an internetwork, the internetwork must be represented in terms of the two basic Nimrod entities: *nodes* and *endpoints*. The internetwork's physical assets, such as routers, point-to-point links, and multiaccess networks, must be captured in Nimrod *maps* comprising interconnected nodes. Traffic sources and destinations must be cast as Nimrod endpoints. Nimrod entities possess attributes (e.g., location with respect to the maps, interconnectivity with other entities, and service information) which are important for routing.

1.1.1 Clustering and Abstraction

Ideally, Nimrod maps should be constructed so as to satisfy the following two primary, and potentially conflicting, goals:

1. Minimize the amount of routing information maintained throughout an internetwork.

2. Maintain routing information sufficient to generate routes that meet traffic service requirements.

To satisfy these goals, Nimrod employs two complementary map construction procedures, namely *clustering* of internetwork physical assets into nodes and *abstraction* of attributes of the component physical assets resulting in node attributes.

The objective of clustering is to reduce the number of entities visible to Nimrod routing. Nodes are usually formed by clustering physical assets possessing similar attributes. These attribute similarities might be in terms of, for example, qualities of service, restrictions on access to services, or ownership of these assets. Such clustering results in a reduction in the amount of information necessary to characterize these physical assets, without a reduction in information detail. However, an internetwork's physical assets may be diverse enough so that clustering according to attribute similarity produces no significant reduction in the number of entities visible to Nimrod routing. In this case, alternative clustering criteria (e.g., geographical locality of physical assets) may be employed.

Clustering may be applied repeatedly, such that physical assets are first clustered into nodes, and then nodes are themselves clustered into larger nodes, and so on. Iterative clustering further reduces the number of entities visible to Nimrod routing and results in a hierarchical organization of nodes with a single top-level *universal* node containing all other entities. In the clustering hierarchy, the clustering criteria applied at different levels may not necessarily be the same.

The objective of abstraction is to reduce the amount of information required to characterize an entity visible to Nimrod routing. Nodes whose component physical assets possess different attributes rely on information abstraction in order to reduce the number of attributes used to characterize them. Abstraction procedures include, for example, eliminating attributes possessed by only a small percentage of the component physical assets or expressing attributes in terms of ranges of values exhibited by these physical assets. Multiple abstraction procedures may be applied to produce the attributes of a given node.

Nimrod does not mandate the choice of clustering and abstraction procedures to invoke in an internetwork. Rather, this choice is a local one under the control of the managers of the various portions of the internetwork, and hence network managers may develop procedures that best suit their needs. We note that the specific clustering and abstraction procedures employed in an internetwork may have a significant effect on the quality of routes generated and on the cost of routing information maintenance. Hence, network managers should exercise care in selecting and using these procedures and may wish to experiment with several different ones before making a final decision. Although clustering and abstraction procedures may be fully automated, we recommend allowing manual intervention in order to enable network managers to make cost-benefit tradeoffs appropriate for their particular networks.

1.2 Nimrod Entities

All of the Nimrod routing functions are performed with respect to an internetwork's representation in terms of the basic Nimrod routing entities, namely nodes and endpoints. Each Nimrod entity has a set of attributes, each of which may be established through one or more of the following methods:

1. Manual configuration.
2. Automatic acquisition during initialization.
3. Active measurement.
4. Abstraction of attributes of component nodes.

A node is a set of contiguous internetwork physical assets. It may be formed by clustering physical assets directly or by clustering existing nodes. If the given node itself comprises component nodes, the routing system employed to route traffic within or across the node is Nimrod routing. Otherwise, this routing system may be any other. A node's attributes include its:

1. *Node identifier* (NID). An NID is a location-independent referent for a node. It is a globally unique, relatively short, fixed-length bit string used by Nimrod-capable devices to communicate node identity (primarily used before a node acquires its locator).
2. *Locator*. A node's locator is a globally unique label describing the node's position in the clustering hierarchy. It consists of the global locator of the node's enclosing node in the clustering hierarchy, concatenated with a local locator, called an *element*, unique among all component nodes of the enclosing node.
3. Pool of locator elements that may be assigned to its component nodes.
4. Constraints on forming *associations* with endpoints. An association is a relationship formed between a node and an endpoint, such that the endpoint acquires a locator from the node. A node may be associated with multiple endpoints, and an endpoint may be associated with multiple nodes.
5. Constraints on forming *adjacencies* with other nodes. An adjacency is a neighbor relationship formed between two nodes that are physically joined. The neighbor relationship need not be symmetric. For example, nodes X and Y may agree to a relationship in which Y is adjacent to X, but X is not adjacent to Y.
6. Maps consisting of the node's current adjacencies and service offerings.
7. Credentials of the node's manager, used in forming node adjacencies and endpoint associations.

An endpoint is a traffic source, destination, or both that is visible to other Nimrod entities through association with one or more Nimrod nodes. Examples of endpoints include hosts and routers or even processes within hosts and routers. An endpoint's attributes include its:

1. *Endpoint identifier* (EID). An EID is a location-independent referent for an endpoint. It is globally unique, relatively short, fixed-length bit string used by Nimrod-capable devices to communicate endpoint identity.

2. *Names.* A endpoint name is a globally unique, variable-length, structured, ASCII string used primarily by humans to refer to the endpoints. Nimrod uses Domain Name System (DNS) names for this purpose.

3. Constraints on forming associations with nodes.

4. Locators. An endpoint's locators are obtained from the nodes with which the endpoint is associated. Therefore, as an endpoint may be associated with more than one node, it may obtain more than one locator.

5. Traffic service requirements from the perspectives of the endpoint as source and as destination.

6. Credentials of the endpoint and its manager, used respectively in authentication of routing information and in forming node associations.

1.3 Nimrod Routing Functions and Databases

At the core of Nimrod lies a set of distributed databases containing routing information that is constructed, accessed, and acted upon by the routing functions. These databases and their relationships to the routing functions are as follows:

1. *Node attributes.* Each node has a set of attributes used in forming node adjacencies and endpoint associations, in constructing maps, and in assigning locators to component nodes.

2. *Endpoint attributes.* Each endpoint has a set of attributes used in forming node associations and in selecting routes.

3. *Endpoint/locator associations.* Nimrod endpoint locators are used in generating routes between and in forwarding messages toward those endpoints. Endpoint/locator associations are stored and accessed through the DNS.

4. *Maps.* Each Nimrod node has a set of maps describing its traffic service offerings and adjacencies to other nodes, collectively called *connectivity specifications* and used in generating routes.

5. *Routes.* Routes are generated in response to requests on behalf of traffic sessions between endpoints. Route generation works within the constraints of the service requirements specified by the endpoints and the services and adjacencies advertised in maps. Each route is expressed in terms of nodes and their corresponding connectivity specifications and is used to construct forwarding information to be installed in routers.

6. *Forwarding information.* Nimrod traffic forwarding is path-oriented, where a *path* is defined by the forwarding state stored in routers according to the route selected and is under the control of the source and destination endpoints.

Databases relevant to but not maintained by Nimrod are the pool of NIDs that may be assigned to nodes and the pools of EIDs and names that may be assigned to endpoints.

Each of the Nimrod routing functions uses portions of the contents of one or more of the Nimrod databases. In a dynamic internetwork, the procedures for updating and retrieving the contents of Nimrod databases will be performed frequently. Therefore, each Nimrod database is organized to optimize the performance of these procedures along the dimensions of delay, internetwork resource consumption, fault tolerance, and load balancing.

Nimrod databases are maintained by a combination of routers, hosts, and special-purpose physical devices. The use of special-purpose devices means that routers and hosts do not have to assume all of the processing and memory load related to routing. For example, as route generation is a computationally intensive procedure, some network managers may elect to use dedicated devices, distinct from routers, whose sole purpose is to generate Nimrod routes.

For most Nimrod databases, we suggest distributing database contents over several physical devices throughout an internetwork. In a large internetwork, one may in fact have no other choice; the memory required for a single Nimrod database may exceed the storage capacity of any one device. Also, we suggest distributing database contents with partial redundancy, such that each database entry is stored in more than one device. Distributed organization of Nimrod database contents helps to reduce the database maintenance and query-response costs borne by any one physical device. Partial redundancy helps to increase the availability of database contents and to reduce the costs of the average database query-response; it may also increase the cost of the average database update, however.

1.3.1 Nimrod Database Consistency

Each Nimrod database contains routing information crucial for successful communication between endpoints. Inconsistencies between the actual state of the internetwork and the state as reflected by Nimrod database contents may result in impaired communication between a pair of endpoints and, in the worst case, may completely disrupt communication among all endpoints. Thus, minimizing the number as well as the consequences of such inconsistencies is a primary objective of the Nimrod database maintenance procedures.

Inconsistencies between database contents and actual internetwork state may result from delays incurred in updating database contents following internetwork state changes. Many of the Nimrod databases are volatile and hence require mechanisms for keeping the contents current, in order to prevent propagation and use of stale routing information. Database maintenance includes rapid and reliable updating with new information as well as removal of old information. We recommend that each Nimrod database be maintained as a cache, such that each entry has a finite lifetime and may be removed from the database when it expires. Cache entry lifetimes will depend upon the expected duration of the usefulness of the cached information.

Inconsistencies between database contents and internetwork state may also result from errors introduced directly into database contents. Errors in Nimrod database contents may be injected inadvertently, through faults in the transmission media or in physical device memory, through misconfiguration, or through incorrect implementation of the database maintenance procedures. Errors may also be injected intentionally by malicious parties, through distribution of fictitious database updates and responses to queries (by capturing and corrupting existing database messages or by generating new messages) or through modification of database maintenance procedures.

Updating and retrieval of Nimrod database contents involve frequent communication of routing information over an internetwork and hence expose this routing information to numerous potential opportunities for error introduction. Therefore, the protocols that carry out these procedures attempt to protect the routing information from introduced errors. In particular, the protocols for communicating information to or from a Nimrod database permit the intended recipient of that information to:

1. Authenticate the information.
2. Detect corruption of the information.
3. Determine whether the information received is newer than any related information the recipient already possesses.
4. Indicate to the sender the receipt of acceptable or unacceptable information.

These protocols also permit the sender to retransmit to the intended recipient any information that it perceives the recipient has failed to receive successfully.

We note that while Nimrod requires consistency between database contents and internetwork state, it does not require different physical devices to maintain identical views of internetwork state: for example, two different routers might maintain different maps for the same node, both of which are consistent with the physical assets of that node. Furthermore, Nimrod does not require consistency in route selection across different physical devices: for example, two different routers might select routes to the same destination such that each router is included in the other's route. The underlying path-oriented nature of message forwarding in Nimrod enables loop-free forwarding in the presence of such inconsistencies in route selection among routers.

1.4 Nimrod Agents

Within an internetwork, each Nimrod database is stored in a set of physical devices. Each physical device containing a portion of a Nimrod database executes a set of functionality, called a Nimrod *agent*, for manipulating the database contents. A single device may contain portions of more than one Nimrod database and hence may contain more than one Nimrod agent. Each Nimrod agent is a Nimrod endpoint.

For each Nimrod database, certain agents are responsible for maintaining specific portions. Such an agent is designated as an *authoritative source* for that portion of the database. A specific portion of a database may have multiple authoritative sources. Each agent is an authoritative source for some portion of a database but may also obtain and cache information learned from authoritative sources for other portions of that same database. In addition to receiving unsolicited database updates, a Nimrod agent may also refresh its database by querying other agents of the same type for their database contents.

Nimrod agents and databases are organized according to the clustering hierarchy, such that each node has a set of agents that act on its behalf to answer or propagate database queries. The node on whose behalf an agent acts is called the agent's *scope*, and a single agent may have multiple scopes.

The Nimrod agents and their corresponding functions are as follows:

1. *Node representatives.* Each Nimrod node must have one or more representatives which maintain the database of the node's attributes and act on its behalf. A node representative is responsible for forming adjacencies with other nodes; forming associations with endpoints; assigning locators to component nodes; collecting maps from component nodes; and constructing its node's maps. Node representatives are authoritative sources for the maps of the nodes they represent.

2. *Endpoint representives.* Each Nimrod endpoint must have one or more representatives which maintain the database of the endpoint's attributes and act on its behalf. An endpoint representative is responsible for forming associations with nodes; discovering locators of the endpoints with which its endpoints wish to communicate; requesting routes to those endpoints and ensuring that the routes satisfy its endpoints' service requirements; initiating path setup along the selected routes; and forwarding data along the established paths. Endpoint representatives are authoritative sources for the locators and service requirements of the endpoints they represent.

3. *Route agents.* Each node may have one or more route agents responsible for collecting maps from nodes throughout the internetwork and for generating and dispensing routes based on endpoint service requirements and node connectivity specificiations advertised in maps. Route agents are authoritative sources for the routes they generate.

4. *Forwarding agents.* Each node must have one or more forwarding agents responsible for initiating neighbor relationships with forwarding agents in other nodes; requesting routes; installing forwarding information in routers; forwarding messages along established paths; and controlling traffic flow into and out of the node according to the node's access restrictions. While each of these functions could be performed by a different type of agent, we have elected to concentrate them in the forwarding agents, in order to minimize the number of different agent types performing the Nimrod routing functions. Forwarding agents are authoritative sources for the portion of the paths that traverse them.

Agents acting on behalf of a node need not reside within that node. Nevertheless, we recommend locating all Nimrod agents (and their databases) close to the entities on whose behalf they act. Such location minimizes delay and internetwork resource consumption when updating the databases corresponding to those entities and in responding to queries from other agents in the vicinity of those entities. A Nimrod agent residing external to the node on whose behalf it acts must be configured with location information for that node, and in some cases for ancestral and descendant nodes as well, in order communicate with other agents that act on behalf of the same node. We recommend placing all agents within the node on whose behalf they act, and henceforth we describe agent behavior from this perspective.

2 Nimrod Operation : An Overview

This section describes key operating procedures in Nimrod. The description is organized based on the pertinent database, in the order given below:

1. Maps: Construction, Dissemination.

2. Routes: Generation, Acquisition.

3. Locators: Acquisition, Notification.

4. Adjacencies: Formation.

5. Paths: Setup, Teardown, Forwarding

6. Discovery: Discovering Neighbors, Agents

This section only provides a brief summary of the operations. For a detailed exposition, the reader is referred to [2].

If this is the .ps version of the document, then the description of most of the operations contains paranthesized references to figures (given in Appendix 1), which may be used for a quick overview of the operation.

2.1 Maps

Each Nimrod node has a set of maps describing its traffic service offerings and its adjacencies. Maps are maintained and updated by node respresentatives. A node representative maintains two kinds of maps for its node: a *basic map* that depicts the child nodes, the arcs interconnecting children nodes and the arcs connecting child nodes to external nodes plus the services provided by each of these nodes and arcs; an *abstract map* that depicts the node, the arcs connecting it to other nodes and the services provided by the node between any pair of incoming and outgoing arcs and by the outgoing arcs themselves. A node representative may construct its abstract map using information obtained from abstraction of basic map, configuration or measurement of service qualities across the node.

Maps are updated using *map updates*, and obtained using *map queries* as described below.

2.1.1 Map Update

A route agent of a node Z that requires the map of another node Y in order to construct its route sends a "map query" to Y. Y responds to Z with its abstract or basic map.

Maps are automatically updated in response to topological changes using constrained hierarchical flooding, according the procedure described below. (If this is the .ps version, you may use the paranthesized references to Figure 1).

The update originates from a node representative of the node (Nr) whose abstract map has changed. This node representative (Nr) sends (1) the new abstract map to one of its boundary forwarding agents (F), who also acts as a forwarding agent at the next higher-level node. That forwarding agent sends (2) the map to a node representative sharing that scope (Nr_p), which in turn distributes (3.x) it to all of the node representatives (Nr_p) and all of the route agents (R_p) sharing that scope.

The change in the abstract map of a node causes a change in the basic map of the enclosing node. This in turn, may or may not cause a change in the abstract map of the enclosing node. If it does, then a designated node representative originates another update to the next higher level by sending (4) the new abstract map to a boundary forwarding agent (F_p). The procedure described above now applies again at this higher level.

In the worst case, a change in a node's abstract map may force changes in all of its ancestral nodes' abstract maps. We expect such changes to be rare, especially in nodes whose descendants are multiply connected.

Map Update is realized using the Update protocol (see section 6)

2.1.2 Map Query

A map query is sent by a route agent wishing to obtain the map of a node for route computation. The kinds of maps that can be requested are as follows:

1. Abstract Maps. Two kinds of abstract maps can be requested - *full* or *abbreviated*. An abstract map is full if it contains service information and abbreviated if it does not.
2. Basic Maps. Two kinds of basic maps can be requested - *complete* or *partial*. A basic map is complete if it contains abstract maps of all component nodes and is partial if it only contains maps of a proper subset of the component nodes.

A route agent sends the map query towards the targeted node (if this is the .ps version, see Appendix 1 -Fig 2 for assistance). When the map query reaches a boundary forwarding agent (F) for the targeted node, this forwarding agent relays (2) the query to a node representative (Nr) for that node. The node representative (Nr) responds (3) to the route agent with the largest subset of the requested map that is consistent with the map distribution restrictions. If it does not have the maps to fulfill the query, or if its restrictions do not permit it to respond, it still sends a reply to the requestor containing a reason for failure.

The Map Query is realized using the Query Response protocol (see section 7).

2.2 Routes

Routes are generated and stored in response to requests on behalf of traffic sessions between endpoints. Each route is expressed in terms of nodes and the connectivity specifications corresponding to these nodes and is used to construct forwarding information to be installed in routers.

Routes are generated using a *route generation* algorithm and stored in the route database. Forwarding agents and endpoint representatives wishing to set up sessions obtain these routes from the route agent using *route requests*.

2.2.1 Route Generation

. The route generation algorithm takes as input a map of the topology, and a set of session service requirements, and the source and the destination node locators. It produces as output a (set of) route(s), or an indication that no route can be found. Each route contains a sequence of node locators and connectivity specification labels for nodes that have to be traversed in order to meet the service requirements.

Route generation algorithms are *not* specified by Nimrod. Rather, each route agent may choose to implement its own route generation algorithm.

2.2.2 Route Request

An endpoint representative or a forwarding agent that receives a transport request and wishes to obtain a route to a destination may request a route from a route agent sharing its scope. Note that if the transport request has no strict traffic service requirements, in terms of quality, monetary cost, or access restrictions, a route may not need to be acquired (the source and destination endpoint locators may suffice for the route). Note also that the route agent to which a route request is sent may not be in the same node as the requestor.

The route request contains the source endpoint's EID and locators, the destination endpoint's EID and locators and the traffic service requirements. In response to a route request, a route agent first searches its route database for a set of feasible routes and if unsuccessfull, invokes the route generation algorithm. The route agent may, in the process of attempting to generate feasible routes, obtain more basic maps of nodes using the map query procedure described in section 2.1.

A route agent responds to the requestor with either a set of feasible routes or an indication that no feasible route could be found.

Route acquisition is realized using the Query Response protocol, described in section 7.

2.3 Locators

Nimrod nodes and endpoints require locators for routing. Each node has exactly one locator and each endpoint is associated with at least one locator. Locators are assigned during initialization following activation, reconfiguration, or movement. The node representatives of a node are responsible for acquiring node locators, and the endpoint representative is responsible for acquiring locators for all of its endpoints.

2.3.1 Acquiring Node Locators

Node locator acquisition involves two phases - first a node representative *acquires* a locator from another node representative (for instance, of the enclosing node). Next it *notifies* all of the node representatives of all descendant nodes of the new locator. The first phase is illustrated in Appendix 1, Figure 4 (only in .ps versions) and the second phase in Figure 5. Readers can use the parenthesized pointers to those figures in the description below to aid in the understanding.

A node representative (Nr) wishing to acquire a locator for a node Z first determines the node S from which its locator should be acquired. This node representative (Nr) then sends (1) a locator request message, containing S, to a boundary forwarding agent (F) which forwards (2) the locator request message to a neighboring boundary forwarding agent, which in turn relays (3) the message to a node representative for S (Nr_p). The request contains information that enables the recipient node representative (Nr_p) to evaluate the request and decide whether it can or wants to honor the request.

If it decides to honor the request, the node representative responds (4) with a new node locator, unique among the locators of S's component nodes. Upon receiving the response, the originating node representative (Nr) decides whether to accept the locator.

If it (Nr) decides to accept the locator, it starts the notification phase (refer Fig. 5) . It notifies all agents in its scope, and all agents in Z's descendant nodes. The latter is done by forwarding the message to each interior forwarding agent (F_c) in Z which is also a boundary forwarding agent for one of Z's component nodes. These forwarding agents distribute the message to all agents in the component nodes including boundary forwarding agents (F_{c_c}) and so on until the locator trickles down to all of the nodes that have Z as an ancestor.

Node locator acquisition is realized by the Query Response protocol (see section 7). Notification to all agents in component nodes is done using the Update protocol (see section 6).

2.3.2 Acquiring Endpoint Locators

An endoint representative attempts to acquire a set of locators for each of its endpoints lacking a preconfigured set of locators. The endpoint representative, say Er, selects a set of target nodes and for each selected node S, sends a locator request message containing S to the node representative for S. As in the case of node locators, if this node representative decides to honor the request, it sends a response containing the locator.

The endpoint locator request and response is realized by the Query Response protocol.

2.4 Adjacencies

One of the attributes of a node is its current adjacencies with neighboring nodes. An adjacency is a neighbor relationship formed between two nodes that are physically joined. The neighbor relationship need not be symmetric, i.e., node A may be adjacent to node B but not vice-versa.

Adjacencies of a node Z to an external node Y are formed by clustering together adjacencies of component nodes of Z to Y. At the lowest level, adjacencies are the physical connections themselves.

2.4.1 Adjacency formation

A single designated node representative is responsible for forming adjacencies between its node and neighboring nodes. When forming adjacencies by clustering existing adjacencies (or physical connections), the node obtains candidate external adjacencies from the node's basic map and groups these adjacencies according to which of their destination nodes are components of the same enclosing node. This information defines the target node for the adjacency formation requests.

For each candidate adjacency, the node representative initiates an adjacency formation procedure (depicted in Appendix 1 - Figure 6 if this is the .ps version, and referred to in the description below). The node representative (Nr) begins by sending an adjacency request (1) to a node representative for the specified node (Nr_s). Using information present in this request, the recipient node representative (Nr_s) determines whether or not to honor the request, and replies (2) to the requesting node representative (Nr) about its decision. If the response is positive, then the node representative decides whether or not to accept the adjacency. If it decides to accept, then it updates (3.x) all of the node representatives and boundary forwarding agents in the node with the newly formed adjacency. It also informs (4) the neighboring node representative (Nr_s) which in turn updates (5.x) all boundary forwarding agents and node representatives in its node.

The acquisition of adjacency is done using the Query-Response protocol (section 7) The distribution of the adjacency update to node representatives and forwarding agents in the nodes forming the adjacency is done using the Update protocol (section 6).

2.5 Paths

Nimrod supports two distinct data message forwarding modes: *flow* and *datagram*. For each mode, a forwarding agent's "next-hop" forwarding decision is dictated by the information stored in its forwarding database and by information contained within the message to be forwarded.

Flow mode requires the establishment of session-specific forwarding state in certain forwarding agents along the routes selected for a traffic session. With flow mode, each session is assigned one or more *paths*, derived from the selected routes. A path corresponds to forwarding state stored in forwarding agents along a route, and each path has a label which is unique within all of these forwarding agents (but not necessarily globally unique.) Distinct traffic sessions may use the same path, and distinct paths may use the same route. The minimum forwarding state required for flow-mode forwarding includes linkages between the path label and the path's previous- and next-hop forwarding agents (and service guarantees for traffic control, if any). In flow mode, data messages carry the path label which guides the message forwarding decisions at forwarding agents along the path.

Datagram mode does not require the establishment of any session-specific forwarding state. In datagram mode, data messages carry a description of the selected route, which guides the message forwarding decisions at forwarding agents along the route. Each forwarding agent at the beginning of a route segment makes an independent forwarding decision for that segment, and hence the session source and destination relinquish some control over message forwarding. However, datagram mode provides robust forwarding, in the sense that the intermediate forwarding agents can base their message forwarding decisions on the current state of their portion of the internetwork.

Both of the Nimrod forwarding modes rely on the existence of underlying paths to fill in route segments. Nimrod can support unicast and multicast paths using the same mechanism. A path may connect a source endpoint to one or more destination endpoints.

Forwarding agents execute a path management procedure to install path state in and remove path state from their forwarding databases. These procedures are described below.

2.5.1 Path Setup

For simplicity of discussion, we focus on unicast paths in the remainder of section 2.5.1. Multicast paths are treated in detail in [4].

Paths may be set up from source to destination or from destination to source. Each path has an *initiator* and a *target*. We expect that most paths will be set up from the source endpoint to the destination endpoint. Hence, the initiator usually begins the path setup procedure on behalf of the source endpoint, and the target usually accepts or rejects a path on behalf of the destination endpoint.

Nimrod paths are inherently multilevel as follows. We begin with a single path, p^1_1, derived from the selected route between the source and destination endpoints for the traffic session. (The superscript indexes the level of the path, where the top level is 1.) This path comprises multiple contiguous paths, ${p^2}_1, \ldots, {p^2}_n$, one for each of the n segments of the route on which p^1 is based. (The subscript indexes the path for the corresponding route segment of the higher level path.) Each ${p^2}_j$ itself comprises multiple contiguous paths corresponding to each of its segments, and so on. In general, for each ${p^i}_j$ composing ${p^{i-1}}_k$, the initiator and target of ${p^i}_j$ maintain linkages to the path ${p^{i-1}}_k$, which helps to guide forwarding along the successive segments of ${p^{i-1}}_k$.

Forwarding agents and endpoint representatives try to form paths by piecing together existing paths rather than by setting up new paths. This method provides the lowest-cost message forwarding in terms of the amount of route generation and forwarding state installation required. In a busy internetwork, there are likely to be many existing paths, and hence we expect this mechanism to be much less expensive than individually setting up and maintaining paths for each traffic session.

We now describe how a new traffic session uses paths at multiple levels, distinguishing the actions in flow mode and datagram mode where appropriate. Whenever the endpoint representative receives a data transport request, it always checks whether there already exists a satisfactory path for the session. This is true whether the new session desires flow or datagram mode forwarding. If a satisfactory path exists, the endpoint representative links the session to the path and forwards session traffic along that path. If no such path exists, however, the endpoint representative attempts to obtain a feasible route for the session. Note that route generation might not be required and that a feasible route might include only the source and destination locators.

After obtaining a feasible route, the endpoint representative proceeds to determine where to install the necessary forwarding state.

Flow mode: The endpoint representative becomes the initiator of a new path, ${p^1}_1$, and generates a path setup message.

If the route contains more than the source and destination locators, the endpoint representative then checks whether there already exists a satisfactory path for the session from itself to the next node in the route specification. Provided such a path, ${p^2}_1$, exists, the endpoint representative proceeds as follows:

Flow mode: The initiator of ${p^2}_1$ (which is also the initiator of ${p^1}_1$) links ${p^1}_1$ and ${p^2}_1$ in its forwarding database and sends ${p^1}_1$'s setup message to the target of ${p^2}_1$. Upon reception of the setup message, the target of ${p^2}_1$ also links ${p^1}_1$ and ${p^2}_1$ in its forwarding database.

Datagram mode: The initiator of ${p^2}_1$ sends the data message to the target of ${p^2}_1$.

If no satisfactory path, ${p^2}_1$, yet exists between the first two nodes in the route specification, the endpoint representative attempts to form such a path by piecing together existing paths. The endpoint representative attempts to find an existing path whose destination locator is the longest match on desired destination's locator and matches the session's *context* (context of a session is the lowest node in the hierarchy that contains the source and destinations). If such a path, ${p^3}_1$, exists, the endpoint representative proceeds as it did with ${p^2}_1$. Otherwise, the endpoint representative proceeds to the enclosing node's enclosing node and again looks for a satisfactory path, and so on.

If the endpoint representative fails to find a satisfactory path to any of the second node's ancestral nodes contained within that context, then there are no "direct" paths to the second node. The endpoint representative then seeks a path up to the first node's enclosing node, as there are likely to be more existing paths between higher-level entities. To this end, the endpoint representative checks whether there already exists a satisfactory path whose target is an exit point of the first node's enclosing node. If such a path, ${p^3}_1$, exists, the endpoint representative proceeds as it did with ${p^2}_1$. Otherwise, the endpoint representative attempts to obtain a feasible route and set up a path, ${p^3}_1$. If there are no short-cut paths from the first entity's enclosing node to the second entity, the above procedure may need to be repeated for successively higher-level ancestors of the first node, up to the context of ${p^2}_1$.

This iterative path formation procedure is performed by the target of each path thus selected, which then becomes the initiator for the path for the next segment, and so. The procedure terminates after attaining the last node in the route, possibly linking together paths at many different levels.

In the presence of multilevel paths, each data message carries a nested sequence of path labels, in order to enable all forwarding agents involved in the paths to forward the message correctly. Intermediate forwarding agents update the path labels in the message, according to the linkages between paths stored in their forwarding databases. Upon receipt of a data message, the target of path ${p^j}_i$ finds it is linked to path ${p^{j-1}}_k$ which in turn is linked to path ${p^j}_{i+1}$ and hence is the initiator of ${p^j}_{i+1}$. This forwarding agent strips off the label for ${p^j}_i$ and replaces it with the label for ${p^j}_{i+1}$, before forwarding the message along that path.

2.5.2 Path Acceptance

Each setup message in flow mode and each data message in datagram mode contains the route specification and additional service requirements, such as resource reservation requests. A boundary forwarding agent or endpoint representative receiving a setup or datagram message determines message acceptability. Acceptability is in part based on the perceived consistency between the route specification and service requirements contained in the message and the service attributes of each node traversed.

When a forwarding agent refuses a setup message, it informs the other forwarding agents on the path between and including itself and the initiator. At the target, once a setup message passes the service attribute consistency check, it must also pass an endpoint-specific consistency check. In particular, the target determines the perceived consistency between the route specification and service requirements contained in the message and the service requirements of the target endpoint. Each target that accepts a setup message informs the initiator. If there is an inconsistency with the target endpoint's service requirements, the target takes one of two actions, depending upon whether the target is the path's destination or source:

1. If the target is at the destination endpoint, it returns to the initiator a message containing its endpoints' destination service requirements. The initiator is then responsible for obtaining a route and setting up a path that is consistent with both the source and destination service requirements.

2. If the target is at the source endpoint, it returns to the initiator a message indicating that it will generate its own route. The target is then responsible for obtaining a route and setting up a path that is consistent with the source service requirements and the destination service requirements contained in the setup message.

Any forwarding agent or endpoint representative may tear down a path by removing the corresponding forwarding state from its forwarding database. Reasons for path teardown might include:

- Detection of a connectivity failure along a path.
- A change in node service attributes or traffic service requirements such that the route on which the path is based is no longer feasible.
- Path expiration if a path exceeds a maximum prescribed lifetime.
- Path preemption in favor of another path.

2.6 Discovery

Information about agents in a node and neighbor information will be configured to start with.

3 Nimrod Databases

3.1 Agent Information

3.1.1 Information Maintained by All Agents

Each Nimrod agent maintains information necessary for inter-agent communication. This information includes the following:

1. Type (e.g., route agent). Must be configured.
2. EID (each agent is a Nimrod endpoint). Must be configured.
3. Scope(s) of agent (i.e., nodes on whose behalf the agent acts). Must be configured.
4. Locator of agent. May be configured or acquired from the enclosing node.
5. Non-Nimrod location information for the agent (e.g., Ethernet address, IP address). May be configured or learned through ARP or dynamic host configuration (DHCP).
6. Multicast address pertaining to all agents of the same type and scope (for networks with multicast capabilities). Must be configured.
7. For each neighboring agent:
   (a) Type. Must be learned through neighbor discovery.
   (b) EID. Must be learned through neighbor discovery.
   (c) Scope(s). Must be learned through neighbor discovery.
   (d) Locator. Must be learned through neighbor discovery.
   (e) Non-Nimrod location information. May be configured or learned through neighbor discovery.
8. For each *key* agent (i.e., node representative, endpoint representative, boundary forwarding agent, and route agent) with the same scope:
   (a) Type. Must be learned through agent discovery.
   (b) EID. Must be learned through agent discovery.
   (c) Scope(s). Must be learned through agent discovery.
   (d) Locator. Must be learned through agent discovery.
   (e) Non-Nimrod location information. May be configured or learned through agent discovery.

3.1.2 Information Maintained by Node Representatives

Each node representative maintains information pertaining to map construction. This information includes the following:

1. Adjacency constraints (implying the set of nodes with which this node may form an adjacency). Must be configured.
2. Association constraints (implying the set of endpoints with which this node may form an association). Must be configured.
3. Set of locator elements that may be assigned to component nodes. Must be configured.
4. Locator. May be configured or acquired from the enclosing node.
5. For each adjacency:
   (a) Node identifier of the adjacent node. Must be learned through adjacency formation.
   (b) Locator of the adjacent node. Must be learned through adjacency formation.
   (c) Component lower-level adjacencies (if such exist). Must be determined during adjacency formation.
   (d) EIDs of boundary forwarding agents participating in the adjacency.
6. Basic map of the node (i.e., the abstract maps of the component nodes). Must be learned from the component nodes.
7. Abstract map(s) of the node. May be configured or constructed from the node's basic map. For each connectivity specification expressed in the map:
   (a) Label.
   (b) Service offerings (e.g., delay, throughput, reliability, security).
   (c) Service restrictions (e.g., who can use the service, when, how, and for what fee).
   (d) Set of node entry and exit points (expressed in terms of adjacent nodes) between which the connectivity specification applies. Transit connectivity specifcations must contain at least one entry and one exit point, incoming connectivity specifications must contain at least one entry and no exit points, and outgoing connectivity specifications must contain at least one exit and no entry points.
   (e) Distribution restrictions (implying the set of nodes to which the connectivity specification should not be distributed).

3.1.3 Information Maintained by Route Agents

Each route agent maintains information necessary for generating maps and responding to route requests. This information includes the following:

1. Basic and abstract maps (from portions of the internetwork). May be configured or acquired from nodes.

2. Routes. May be configured or generated from maps and traffic service requirements specified in route requests. For each route:
   (a) Locator of each constituent node.
   (b) Label of connectivity specification for each constituent node.
   (c) Summary of the route's offered services and restrictions.

3.1.4 Information Maintained by Forwarding Agents

Each forwarding agent maintains information necessary for forwarding Nimrod messages. This information includes the following:

1. Routes (maintained by path initiators). May be configured or acquired through route requests.
2. Paths (maintained by path initiators). May be configured or constructed based on path requests. For each path:
   (a) Path label.
   (b) Requested services.
   (c) Route.
3. Forwarding information. May be configured or constructed from paths. For each incoming path:
   (a) Incoming path label and actions (e.g., terminate path, which causes the next higher-level path label in the message to be treated as an incoming path).
   (b) Previous hop agent EID and non-Nimrod location information.
   (c) Outgoing path label(s) and actions (e.g., insert a stack of path labels into the message so that it can be forwarded).
   (d) Next hop agent EID and non-Nimrod location information.
4. For a boundary forwarding agent:
   (a) Traffic handling restrictions (i.e., connectivity specifications used in determining which traffic may enter and leave the node). May be configured or learned from the node.
   (b) For each of the node's adjacencies, the set of boundary forwarding agents participating in the adjacency (used in forwarding traffic across the node). Must be learned from the node.

3.1.5 Information Maintained by Endpoint Representatives

Each endpoint representative maintains information pertaining to the endpoints on whose behalf it acts. (In a future version of Nimrod, some of this information may be dynamically acquired from the endpoints themselves, but the initial version of Nimrod contains no protocols for passing such information between endpoints and their representatives.) For each endpoint, this information includes the following:

1. EID. Must be configured.

2. DNS name(s). Must be configured.
3. Non-Nimrod location information (e.g., IP address). Must be configured.
4. Association constraints (implying the set of node with which this endpoint may form an association). Must be configured.
5. Service requirements of endpoint traffic (from both the source and destination perspective). Must be configured. This information includes:
   (a) Service requests (e.g., delay, throughput, reliability, security).
   (b) Service restrictions (e.g., provider preferences, what the user is willing to pay).
6. Routes. May be configured or acquired through route requests.
7. Paths. May be configured or acquired through path requests.

3.2 Database Specification

Each database is composed of one or more tables. The content of tables can be configured, or can be writable by one or more agents. Tables can be read by one or more agents. Databases can be either private or public. Private databases are the concern of only one Nimrod agent, and, in principle, need not have a standard form. Public databases are those that can be accessed by other than the agent that writes them. Public databases need to have a standard form (or at least the representation of the items "in the wire" need to be standardized). Private databases are described with BBN's Nimrod implementation in mind which assumes a Nimrod router: i.e., a box that can act as a node by itself and that includes the complete Nimrod functionality.

The description of a database includes:

- a brief description of its function;
- where the database is located;
- how the data is going into the database is generated (Producer); this can be the responsibility of an agent or could be via configuration;
- which agent(s) uses the data (Consumer);
- the tables that are part of it.

3.2.1 Adjacency Database

Information required to determine what adjacencies can be created. From this database, a router can infer on a per interface basis which announcements should be sent and what announcements are expected.

Location: At each Nimrod router.

Producer: Configured. (Could also be filled out by management or the Node's representative.)

Consumer: Private to the router.

Adjacencies Table: Information about adjacencies.

1. Self scope: a scope for which the box is acting.
2. Interface: the box interface on which this adjacency is established.
3. Neighbor scope: the scope of the neighbor node with which this adjacency is established.
4. Direction: specifies if the adjacency is incoming or outgoing.
5. Authentication information: information to authenticate outgoing information.
6. Neighbor Authentication Information: information to validate the neighbor's information.
7. Status: specifies whether or not the adjacency is up. (Valid values to be defined.)

3.2.2 End-to-End Sessions Database

Keeps information about active sessions originating or terminating at the particular endpoint representative.

Location: At each router.

Producer: An endpoint representative.

Consumer: The same endpoint representative. (This is a private database.)

Active Session Table: Information on currently active sessions.

1. Initiator: local or peer.
2. Local endpoint locator.
3. Local endpoint address. (IP Address/MAC address).
4. Peer endpoint locator. (This locator might include an IP Address and Mac address.)
5. Peer endpoint Fully Qualified Domain Name (FQDN).
6. Outgoing Traffic Service Requirements
7. Outgoing Traffic Service Restrictions.
8. Incoming Traffic Service Requirements.
9. Incoming Traffic Service Restrictions.

3.2.3 Endpoint Database

Associates non-nimrod information about a host (IP address, etc.) with its assigned Nimrod identity as an endpoint.

Location: At each router that acts as an endpoint representative.

Producer: Configured.

Consumer: The endpoint representative.

Endpoint Table: The first three items identify the endpoint using non-Nimrod information. The last three items are Nimrod information associated with the endpoint.

1. Endpoint FQDN;
2. Endpoint IP Address;
3. Endpoint Mac Address;
4. Endpoint Assigned Scope. Configured.
5. Endpoint Assigned EID. Configured.
6. Endpoint Locator.

3.2.4 Locator Database

Keeps track of pool of available locators and of assigned locators.

Location: At a node's representative

Producer: Node Representative

Consumer: Node representative

Element Availability Table: Available Elements. A one row table

1. Global prefix for node's locators.
2. Element Size: an integer number representing the number of bits in the elements for this node.
3. An integer number smaller than $2^{(ElementSize)}$ representing the next element to be assigned. After an element is assigned, this entry is increased by one.

Assigned Elements Table: Contains the association between assigned elements and the permanent ID of the assignee.

1. Element Value
2. Scope of assignee (can be an EID).
3. When assigned (timestamp).
4. Valid until (timestamp).

3.2.5 Map Database

An XDR specification of maps is defined in Appendix 3. This section specifies the maps kept by a node representative.

Location: At a node's representative.

Producer: For a node made out of component nodes, the basic map is obtained by assembling the maps distributed by component nodes. Otherwise, the basic node is configured. Other maps are derived from the basic map, but their production is outside this specification.

Consumer: Routing agents.

Map Table Table containing maps representing a node.

1. Map type
2. Map number
3. Map

3.2.6 Map Filtering Database

This database specifies which requestors can obtain which maps.

Details to be defined.

3.2.7 Route Database

To be defined.

4 Reliable Transaction Protocol

Many Nimrod protocol messages (such as messages in Update protocol, Query-Response protocol) and other agent transactions require *reliable* delivery. Rather than have each agent duplicate this reliability functionality, Nimrod includes the Reliable Transaction Protocol, which provides its clients the ability to reliably communicate arbitrary size blocks of information between a client and a peer and to receive an arbitrary size reply in approximately one round-trip time.

The reliable transaction service is built on Transaction-TCP (TTCP) (RFC-1644, RFC-1379). TTCP is a backwards-compatible extension of TCP, which is optimized for request/response interactions. In particular, TTCP bypasses the normal three-way handshake required at TCP connection setup time. This bypass is accomplished by adding a "connection count" option in the TCP header, and by maintaining per-client connection history at the server. This information allows the server to correctly distinguish a new connection open (SYN, no ACK) from a duplicate or out-of-order open, without shaking hands with the client. Using TTCP, the client can obtain a response to a request message in one round-trip-time to the server and back (plus the server's processing time).

Sites with TCP source code can update it to perform TTCP by using a set of changes available from ISI in the public domain. The approach of using TTCP for Nimrod transactions is believed to be less effort than adapting a different transaction protocol, or writing a new one.

Transaction TCP is accessed from applications via BSD sockets. For an example, please refer Appendix 2.

4.1 Services Offered

The reliable transaction service permits one or more transactions to be invoked at a peer. The service is offered through a function, defined as:

```
int nim_xact(
  const      uint           flags;
#define NIM_XFLG_NFRE 0x               /* Caller retains xact_datap memory */
  const struct sockaddr      *srcp,           /* Source info: EID, Locator, port */
  const               *dstp;          /* Destination info: EID, Locator, port *
  const uchar         *keyp;          /* Destination centric keying info */
  const char          *serp;          /* Service requirements */
  const struct timeval       *top;            /* Interval before prodding client */
  const int           (*callbackp()); /* To pass reply to client */
      uint             cookie;        /* To identify reply to client */
      uint            xact_len;       /* Length of transaction data */
      NimXact         *xact_datap;    /* Transaction data */
)                                     /* returns 0 if ok, else error
```

Transactions are packed one after another. The beginning of each transaction is the NimXact structure. The NimXact structure is defined in [5], section 5.

When the reply is complete, the transaction callback function is called.

A transaction may be aborted at any time by calling a function defined as:

```
int nim_xabort(
      uint             cookie;        /* Client supplied cookie */
              /* Following set if something was received */
      uint            *flagsp;        /* Control flags */
      uint            *rep_lenp;      /* Reply length, 0 if timeout */
      NimXact         **rep_datapp;   /* Reply data */
)
```

Note that races are possible with nim_xabort; the aborted opertion may have actually been performed by the peer, whether or not any information is returned in the area pointed to by rep_datapp. There is no rollback or CCR facility provided by the reliable transaction service.

5 Path Management Protocol

Nimrod forwarding agents and endpoint representatives are responsible for establishing forwarding state information in routers and hosts, respectively. They are also responsible for forwarding Nimrod messages according to this state information and according to forwarding directives carried along in the messages. The forwarding state information installed and maintained by forwarding agents and endpoint representatives is derived from the route selected and is called a *path*. A path may connect one or more source endpoints and one or more destination endpoints. (For the initial implementation of Nimrod, each multicast path is either a source tree or a sink tree.) Multiple traffic sessions may use the same path, thus helping to contain the amount of internetwork resources consumed in managing paths. Moreover, multiple paths may be established based on the same route.

Forwarding agents and endpoint representatives use the path management protocol to install and remove forwarding state information from their forwarding databases. Each forwarding agent and endpoint representative maintains forwarding information for those paths that originate, terminate, or pass through it (in the case of forwarding agents only).

Paths may be set up from source to destination or from destination to source. Each path has an *initiator* and a *target*. We expect that most paths will be set up from the source endpoint to the destination endpoint. Hence, the initiator usually begins the path setup procedure on behalf of the source endpoint, and the target usually accepts or rejects a path on behalf of the destination endpoint. Forwarding state is established such that messages may be forwarded in either direction of the path.

Forwarding agents and endpoint representatives try to form new paths by piecing together existing paths rather than by setting up entirely new paths, provided the existing paths meet the traffic service requirements. Hence, Nimrod paths are inherently multilevel. This method provides the lowest-cost message forwarding in terms of the amount of route generation and forwarding state installation required. In a busy internetwork, there are likely to be many existing paths, and hence we expect this mechanism to be much less expensive than individually setting up and maintaining the paths required for every traffic session.

Paths are identified by path labels, which are unique along the path but not necessarily globally unique throughout the internetwork. The labels for each direction of a path are distinguished by a bit that indicates whether the direction is toward the target or toward the initiator. By eliminating the requirement for global uniqueness of path labels, we can assign paths relatively short labels, thus reducing the overhead in carrying them in packets and the overhead of looking up forwarding information indexed by these labels. We must be able to assign path labels in a way that reduces the probability of collisions of labels from different paths. Random assignment of path labels will help reduce collisions, but it will not guarantee freedom from collisions. We discuss two candidate path label collision resolution strategies in section 5.2. Note that multiple non-intersecting paths may carry the same labels.

Flow mode data messages contain path labels and user data. Most paths are composed of paths which in turn are composed of paths and so on. Hence, a flow mode data message must contain the path labels of all the component paths at all levels, but only one path label is used for forwarding at a given time. These path labels are stacked in the message and manipulated (pushed and popped) by the forwarding agents and endpoint representatives handling the message.

5.1 Protocol Messages

The path management protocol uses three types of messages: *setup*, *accept*, and *teardown*. Message contents are described below, but explicit formats are not yet provided in this document. All such messages travel along the path to which they refer. Path management protocol messages may be used to collect and return performance monitoring information for a path (e.g., path delay and throughput) as well as to set up and tear down a path. Monitoring operates in two modes: *collection* and *return*. Hence, monitored information may appear in a message as information being collected or information being returned. Moreover, a message may simultaneously collect and return information for different directions of a path.

Setup. The setup message is generated by the path initiator and is used to establish forwarding state in endpoint representatives and forwarding agents. It contains:

1. The label of the path to set up.
2. The route, in terms of the locators of the nodes through which it passes (must at least include the locators of the source and destination nodes) and the label of the connectivity specification used across each node in the route.
3. The EIDs of the source endpoint and the source endpoint's representative.
4. The EIDs of the destination endpoint and the destination endpoint's representative (for unicast).
5. The multicast group identifier for the destinations (for multicast).
6. Indication of whether the path is source- or destination-initiated.
7. Service requirements (expressed as type, length, and value).
8. Monitored information for the path, updated at each node in the path (expressed as type, length, and value).

Accept. The accept message is generated by the path target and is used to indicate successful path establishment from initiator to target. It contains:

1. The label of the accepted path.
2. Any monitored information for the path, collected during setup (expressed as type, length, and value). Accept messages travel backwards along paths. A path may be used for data transport before an accept message is generated by the target or received by the initiator. Note that a source as initiator may wish to wait for an accept message from the target before sending data on a path. For example, if the source pays for all data messages sent, whether or not they are successfully received at the destination, it may want to wait to make sure that the path is successfully established before sending data to the destination.

Teardown. The teardown message is generated by any endpoint representative or forwarding agent on the path and is used to remove forwarding state. It contains:

1. The label of the path to be torn down.
2. The reason for the teardown (expressed as type, length, and value).
3. Any monitored information for the path collected along the path (expressed as type, length, and value). Teardown messages may travel in both directions along paths and may result from any of the following:
   (a) Loss of a lower-level path which is a component of the specified path. Each agent detecting the loss sends a teardown message toward either the initiator or the target, depending on which side of the lost path the agent resides.

(b) A timeout. Paths have a maximum lifetime to ensure that forwarding state for broken paths is eventually removed. The agent detecting the timeout sends a teardown message toward the initiator and toward the target.

(c) A change in service requirements for the traffic session. If the change occurs at the initiator, the initiator sends a teardown message toward the target. If the change occurs at the target, the target sends a teardown message toward the initiator.

(d) Insufficient resources available at the next hop. The agent detecting the lack of resources sends a teardown message toward the initiator. In this case, the agent receiving this message, may try a different next-hop forwarding agent, if one is available. Only if it is unsuccessful at finding a next-hop forwarding agent does the agent receiving the teardown message propagate it back to the initiator.

(e) A change in connectivity specifications for a node on the route. Note that the path should only be torn down if the new connectivity specification conflicts with the service requirements for the path. To make such a consistency check between the path service requirements and the node's connectivity specifications, a forwarding agent must have knowledge of the service requirements for each path. (For the initial version of Nimrod, intermediate forwarding agents will retain the setup message for each active path.) In this case, the forwarding agent sends a teardown message toward the initiator and toward the target.

(f) Unresolvable path label collision. The agent unable to resolve the collision sends a teardown message back toward the initiator.

(g) Preemption in favor of another path. Forwarding agents and endpoint representatives are free to implement their own preemption criteria, but these are not part of Nimrod. (In the initial version of Nimrod, all paths will be set up on a first-come, first-served basis, with no preemption.) In this case, the preempting agent sends a teardown message toward the initiator and toward the target.

All path management protocol messages are covered by integrity and authentication checks and are sent using the reliable transaction protocol between successive hops along the path. This helps to reduce the amount of network resources consumed by retransmissions in lossy environments and may also help to determine problems for unsuccessful message transmissions.

5.2 Protocol Finite-State Machines

There are two finite-state machines for the path management protocol, one applicable to the initiator and one applicable to the target and any intermediate forwarding agent in the path.

Initiator. The state machine for the initiator has four states: *idle*, *check*, *ready*, and *done*. State transitions are depicted below:

- idle → check: This transition occurs when the initiator begins to set up a path.
- check → ready: This transition occurs after the initiator has successfully completed all of the resource availability checks for the path (described later on). At this point, the initiator installs the forwarding information for the path in its forwarding database. From the perspective of the initiator, the path may be used to carry data traffic.

- check → idle: This transition occurs if the initiator fails to complete the resource availability checks for the path.
- ready → done: This transition occurs when the initiator receives an accept message from the target.
- ready → idle, done → idle: This transition occurs when the initiator receives or generates a teardown message for the path. At this point, the initiator removes the forwarding information for the path from its forwarding database.

Receipt of unexpected messages (such as an accept following a teardown) in each state do not cause state changes or issuing of messages, but these anomalous events should be logged for network management purposes.

Intermediate Forwarding Agents and Target. The state machine for the intermediate forwarding agents and the target has three states: *idle*, *check*, and *ready*. State transitions are depicted below:

- idle → check: This transition occurs when one of these agents receives a setup message.
- check → ready: This transition occurs after one of these agents has successfully completed all of the consistency and resource availability checks for the path (described later on). At this point, the agent installs the forwarding information for the path in its forwarding database. From the agent's perspective, the path may be used to carry data traffic.
- check → idle: This transition occurs if one of these agents fails to complete the consistency and resource availability checks for the path.
- ready → idle: This transition occurs when one of these agents receives or generates a teardown message for the path. At this point, the agent removes the forwarding information for the path from its forwarding database. A teardown message received from an upstream neighbor and indicating insufficient resources (such messages are only generated during path setup) may result in the agent resending the setup message to another next hop, if one is available. Only if there are no suitable next hops does the agent propagate the teardown message toward the initiator.

Receipt of unexpected messages (such as an accept following a teardown) in each state do not cause state changes or issuing of messages, but these anomalous events should be logged for network management purposes.

Check State Actions. When in the check state, a forwarding agent or endpoint representative must perform a series of tests to determine whether to install forwarding information for the path. These tests are partitioned into two sets, those related to consistency and those related to resource availability. Consistency checks include verification of the following:

1. The setup message is not out of date (performed the intermediate forwarding agents and the target). If the setup message is out of date, the agent sends a teardown message to the initiator, indicating the reason for the teardown.

2. The setup message is not a duplicate (performed by the intermediate forwarding agents and the target). If the setup message is a duplicate, the agent sends a teardown message to the initiator, indicating the reason for the teardown.

3. The path label carried in the setup packet is not already in use (performed by intermediate forwarding agents and targets that are not endpoint representatives for the session). There are currently two candidate mechanisms:

   (a) If the path label is in use, the agent checks to see whether it can distinguish the new path from the already existing path by using the stack of path labels carried in the setup message for the colliding paths. Provided these path stacks are different, the agent can resolve the path label collision, using stacked labels to distinguish between the paths (Note that this is similar to searching through hash buckets.) If the agent cannot resolve the collision, it sends a teardown message to the initiator, indicating the reason for the teardown.

   (b) If the path label is in use, the agent sends a teardown message to the previous hop in the direction of the initiator, indicating the path label to use. The previous hop may itself have a path collision with this label and may need to iterate to converge on a mutually agreeable label. If the agents cannot agree within a specified small number of iterations, the previous hop agent sends a teardown message to the initiator, indicatrion the reason for the teardon.

4. The following checks are performed by the intermediate forwarding agents only:

   (a) The forwarding agent acts on behalf of the current node in the route specification carried in the setup message. If the forwarding agent does not act on behalf of the node, it sends a teardown message to the initiator, indicating the reason for the teardown.

   (b) The connectivity specification label for the node, carried in the setup message, is a valid connectivity specification for this node. If the connectivity specification is not valid, the agent sends a teardown message to the initiator, indicating the reason for the teardown.

   (c) The node's service restrictions do not preclude carrying traffic along the specified route. If the node cannot carry this traffic, the agent sends a teardown message to the initiator, indicating the reason for the teardown.

5. The route specification carried in the setup message meets the target endpoint's service requirements (performed by the target only). If so, the target sends an accept message to the initiator. Otherwise, the target sends a teardown message to the initiator. indicating the reason for the teardown, and takes one of the following actions:

   (a) If the target is at the destination endpoint, the teardown message contains the destination endpoint's service requirements. The initiator is responsible for obtaining a feasible route that accounts for these service requirements.

   (b) If the target is at the source endpoint, the teardown message indicates that the source will generate a feasible route.

Resource availability checks including verification of the following:

1. The forwarding database can accommodate state for a new path (performed by the initiator, target, and intermediate forwarding agents). If there is insufficient room in the forwarding

database, the agent sends a teardown message to the previous hop in the direction of the initiator, indicating the reason for the teardown.

2. There exists a feasible path to the next node on the specified route. (This check is extensive and may require additional paths to be set up; it is performed by the initiator and the intermediate forwarding agents.) The initiator or intermediate forwarding agent may have to request a route and set up this path, or there may be such a path already established.

   (a) If such a path Q has already been established, the initiator or intermediate forwarding agent links the current higher-level path P with Q so that traffic along P will be forwarded along the lower-level path Q.

   (b) If no such path Q yet exists, the initiator or intermediate forwarding agent obtains a route for the path. This route must go to the next node in P's route, and it must satisfy the session's service requirements.

   (c) If the initiator or intermediate forwarding agent obtains a feasible route, it proceeds to set up the path Qn for that route and determines whether the necessary resources can be reserved along that path. Only when the initiator or intermediate forwarding agent receives an accept message for path Q does it make the corresponding linkages between P and Q in its forwarding database. Note that an intermediate forwarding agent also makes a path label linkage for the reverse direction of the paths P and Q so that messages can travel in either direction along a path.

   (d) If an intermediate forwarding agent fails to obtain a feasible route or if the setting up of Q fails, that agent sends a teardown message to the previous hop in the direction of the initiator of P, indicating the reason for the teardown.

5.3 Forwarding Information in Nimrod Messages

Datagram messages and setup messages for flow mode carry almost exactly the same information and are processed by forwarding agents in similar ways. The principal difference in message processing for datagram and setup messages is that no forwarding state is established specific to datagrams. Most of the consistency and resource availability checks previously described for setup messages are also performed for datagram messages. Shared contents of setup and datagram messages include:

1. EIDs for the source and destination endpoints and their respective representatives.

2. The route in terms of the locators for nodes and their relevant connectivity specifications.

3. Service requirements (e.g., bandwidth guarantees, delay bounds) from the perspective of the initiator of the message. Forwarding agents use this information in deciding how to forward the packet toward the next node on the route.

4. User data (setup messages might not always contain user data).

5.3.1 IPv6 Optimizations

For the initial implementations of Nimrod, we will encapsulate Nimrod packets within IP packets, independent of the version of IP currently in use. Hence, there is no information that must be added to IP packets (other than the identifier of the enclosed protocol, namely Nimrod). All Nimrod-specific information will be carried in Nimrod packets encapsulated within IP.

For an IPv6 internetwork, one may wish to define Nimrod-specific options that will allow some Nimrod information to migrate up to the IP header. The proposed options include hop-by-hop options, a route option, and end-to-end options. Candidate message formats were originally presented at Nimrod Working Group meeting at the December 1994 IETF. We have not elected to refine these formats at this time, because we are not planning on carrying Nimrod information in the IPv6 header in our initial implementation. Should the demand arise, we will revisit this issue.

Hop-by-Hop Options. The hop-by-hop options are those that must be modified at most forwarding agents and include the stack of path labels, carried in flow-mode data messages, and monitoring information, which may be carried in flow-mode data messages or path management messages. The monitoring option enables multiple path performance quantities, such as delay and throughput, to be monitored and updated at each hop along the path. All monitoring information is expressed in terms of type, length, and value. Collected monitoring information is returned to the other end of the path in an end-to-end performance option described below.

Route Option. The Nimrod route specified in terms of locators of component nodes and labels of connectivity specifications would be included in a route option. This information is carried in datagram messages and in setup messages. The format of the route specification follows that described previously for the setup message.

End-to-End Options. Source and destination endpoint identifiers and source and destination endpoint representative EIDs is carried in an end-to-end option. Path service requirements and monitored performance information are carried in separate end-to-end options. Endpoint identifiers and path service requirements are included in datagram messages and in setup messages. Monitored information may be included in flow-mode data messages and in path management messages. Each service requirement and each performance parameter is specified in terms of type, length, and value.

6 The Hierarchical Update Protocol

The Update Protocol is used to update database contents (e.g., the map database) in an efficient manner. By efficient, we mean that the implicit flooding constituting the protocol is carefully constrained by involving only a few agents per update.

The Update Protocol consists of an Update Message that is generated by the agent wishing to make an update to a particular database. The originating agent forwards the message to one or more agents which further forward it to other agents and so on, until all the necessary database locations have been updated. Once an originating or transit agent has successfully forwarded a message, it does not retain any state corresponding to the message. The Update Message uses the Reliable Transport Protocol (see section 4) which provides hop-by-hop reliablity, but no effort is made to ensure end-to-end reliablity. We believe that the resource and complexity overhead demanded by an end-to-end reliablity mechanism does not justify its importance for database updates (note that Nimrod does not absolutely require database consistency (see section 1.3.1). Thus, no acknowledgements (positive or negative) are part of the Update Protocol. Exceptions are handled by triggering SNMP events.

Thus, the state machine constituting the protocol engine at the agents is trivial. However, the decision of whether to cache the message contents or not, and the decision of who to forward the message to requires some specification. We have used an Update Message Forwarding Table (UMFT) for this purpose. There is one UMFT for each of the four types of agents. The UMFT describes the actions to be taken on the receipt of a particular message from a particular agent. The UMFT depends on what functionality we map into the protocol and how we do the mapping.

Currently, the Update Protocol is used for map updates (section 2.1), locator notifications(section 2.3), and adjacency distributions (section 2.4).We note that the UMFT-based protocol specification approach provides flexibility by making it easy to change the functionality and the mapping - you just need to alter the entries in the table. The Update Protocol is typically used by Nimrod agents or other "users" in order to initiate updates. We use the term *client* to denote such users.

The Update Protocol consists of one message, the Update Message (UpdMsg). The UpdMsg consists of a variable length header (NimUpdHdr) followed by database specific information. In this section, we only describe the UpdHdr. Database specific packet formats are specified in section 3.2.

6.1 Update Header

Each item in the common update header is given below along with either the number of bits in the packet used by the item or a reference to a variable length packet format defined elsewhere.

```
NimUpdHdr ::=
{
   org-agent : 8,
   dst-agent : 8,

   flags :16,
   db-type  : 8,

   db-seq :24,
   dst-scop :NimScop,
   org-eid :NimEID,
   org-loc :NimLoc,
   sign :NimSig
}
```

The packet field semantics are as follows.

- **org-agent.** The type of agent originating the update. Agent types are defined in [5].
- **dst-agent.** The type of agents for whom the update is intended.
- **flags.** Miscellaneous operation dependent flags.

- **db-type.** The type of the database that is being updated. At most one kind of database can be updated with an update message. Database types are defined in [5].
- **db-seq**. A 24-bit number that denotes the sequence number of the message with respect to the originating agent EID. That is, the sequence number and the EID together identify the packet uniquely, modulo wraparounds.
- **dst-scop**. The scope of the destination. The update is not to propagate out of this scope. The format of NimScop is given in [5].
- **org-eid**. EID of the agent that issued the update. The format of NimEID is given in [5].
- **org-loc**. Locator of the agent that issued the update. The format of NimLoc is given in [5].
- **Sign**. Authentication field. Contains the signature of the agent originating the update. The format of NimSig is given in [5].

6.2 Agent Protocol Engines

An UpdMsg is originated by a single agent (*originating agent*) and handled by a number of intermediate agents (*transit agent*) enroute to the destination(s). We give briefly the algorithm executed at transit and originating agents.

6.2.1 Transit Agent

Given below is the algorithm executed by a transit agent (ThisAgent) upon receipt of Update Message (UpdMsg) from another agent (PrevAgent). The algorithm uses the UMFT which is described in section 6.3. Forwarding is done using the Reliable Transaction Protocol (section 4).

```
begin
   /* Check message contents. More checks than those given here are probably
      needed. */

    if seq_num(UpdMsg) is invalid    /* duplicate */
    then trigger SNMP event and discard UpdMsg.

    /* Consult the UMFT table for caching and forwarding directions */

    NextAgentType = UMFT(ThisAgent, PrevAgent, UpdMsg)

    if NextAgentType is not a valid agent, discard message.

    repeat
       choose a (previously unchosen) agent X of NextAgentType.
       if none found
       then trigger SNMP event and discard message
       else forward UpdMsg to X.
    until successfully forwarded or message discarded.

end.
```

6.2.2 Originating Agent

Given below is the algorithm executed by an originating agent (ThisAgent) upon receipt of Update Message (UpdMsg) from the Client. The algorithm uses the UMFT which is described in section 6.3.

```
begin

    insert sequence number into UpdMsg

    /* Consult the UMFT table for caching and forwarding directions */

    NextAgentType = UMFT(ThisAgent, Client, UpdMsg)

    if NextAgentType is not a valid agent, discard message.

    repeat
       choose a (previously unchosen) agent X of NextAgentType.
       if none found
       then trigger SNMP event and discard message
       else forward UpdMsg to X.
    until successfully forwarded or message discarded.

end.
```

6.3 The Update Message Forwarding Table (UMFT)

For each of the four agent types (Forwarding Agent, Endpoint Rep., Node Rep., and Route Agent), we give below the actions upon receipt of an UpdMsg. The agents are represented by F, Er, Nr, and R respectively and suffixed by p, c or s to indicate the agent in the parent, child or sibling node of the node under consideration. No suffix means the agent belongs to the node under consideration.

For each agent X, the UMFT consists of 4 columns, details of which are as follows.

1. **Database type**. The db-type in message received by X.
2. **PrevAgent**. The agent from which the message came at the level of THIS protocol. Note that the message may have hopped through many intermediate agents, but this is like a "tunnel" for this protocol. If the message is originated at this agent, then there is actually no PrevAgent, but we use PrevAgent = Client to denote this.

   Note that a boundary forwarding agent for a given node acts also as an interior forwarding agent for its parent node. An unsubscripted F always represents the *boundary* forwarding agent and F_c is an *interior* forwarding agent for the node in question. Sometimes, to emphasize the "interior-ness" or "boundary-ness" of a forwarding agent, we shall prefix by an 'i' or a 'b'. The reader should understand that the following equivalences hold: bF = iF_p = F; iF = bF_c = F_c.
3. **Actions**. This says what agent X will do when the message is received from PrevAgent. Sometimes uses function "forward" whose semantics are described following the table. Com-

binations not given represent "Unexpected" situations. If these happen, an SNMP event is triggered and the message is discarded.

4. **Fig,Ref.** Should be ignored for ascii (.txt) version. For postscript(.ps) version, references a pictorial representation of the action. The figure number y is identified as F-y. In the figure, messages from one agent to another are represented by labelled arrows. In the UMFT, the relevant incoming arrows to X are denoted by Iarw <label1>, <label2> ... and the relevant outgoing arrows to X by Oarw <label1>, <label2> etc. Labels x and -x denote mutually exclusive actions (e.g., Oarw -x is done if Oarw x is not possible).

```
                           UMFT for F
                         --------------
                         **************
===========================================================================
  DB Type      PrevAgent            Actions                      Fig, Ref
===========================================================================
   Map          Nr                                           F-1 Iarw 1
                               forward (UpdMsg, Nrp)............Oarw 2
---------------------------------------------------------------------------
  Locator       Nrp                                          F-5 Iarw 2.2
                               forward (UpdMsg, Nr).............Oarw 3
                                                             (DOWN 1 LEVEL)
---------------------------------------------------------------------------


                            UMFT for Nr
                            -----------
                            ***********
============================================================================
  DB Type      PrevAgent            Actions                         Fig Ref
============================================================================
   Map           Fc                                                 F-1 Iarw 2
                               forward (UpdMsg, {all Nr,R})....... Oarw 3.x
-----------------------------------------------------------------------------
   Map          Client         make new UpdMsg(t=1) with AbsMp;    F-1
(New AbsMap)                   forward (UpdMsg, F)................ Oarw 1
----------------------------------------------------------------------------
   Map           Nr            Cache, ie, Update Map Database      F-1
                                                                   Iarw 3.x
----------------------------------------------------------------------------
  Locator       Client         make UpdMsg with LocAcqrd           F-5
(New Loc Acqrd)                forward (UpdMsg, {all Nr, Er, R,
                                                  A, F, Fc})....Oarw 2.x
----------------------------------------------------------------------------
  Locator        F                                                 F-5
                               forward (UpdMsg, {all Nr, Er, R,
                                                  A, F, Fc})....Oarw 4.x
```

```
                                                             (DOWN 1 LEVEL)
---------------------------------------------------------------------------



                                 UMFT for R
                                 -----------
                                 ***********
============================================================================
  DB Type    PrevAgent           Actions                         Fig,Ref
============================================================================
    Map      Nr              Cache, ie, Update Map DB      F-1 Iarw 3.x
---------------------------------------------------------------------------


Semantics of functions:

  forward (Msg, AgentSet) :
        forward Msg *reliably* to one agent of each type given in Agent Set.
        Includes the functionality of resending to another agent of same type
        an agent cannot be reached. The set {all X} indicates all agents of
        type X including itself. May involve multiple hops. If an agent A
in AgentSet is not 1 hop away, Msg is handed over to a neighboring
forwarding agent that knows how to reach A.
```

7 The Hierarchical Query-Response Protocol

The Q-R Protocol is used to obtain database information(e.g., the map database) in an efficient manner.

The Q-R protocol consists of a Query Message that is generated by the agent wishing to make a query. The originating agent forwards the message to one or more agents and awaits a response to the query. The response to the query is contained in a Response Message that is generated by the agent that has access to the information requested in the query.

If the originator does not get a response within a certain time t, it informs the application of a query failure. The value of t may be configured or given to the protocol by the application. The application also has the option of requesting an abort of the query - in this case, the state is reset and the response is ignored. As in the case of the Update Protocol, exceptions are handled by triggering SNMP events.

The state machine for the originating agent has two states - INITIAL and WAITING (for a response). A transition from INITIAL to WAITING is made when a query is sent and the state is reset to INITIAL when a response arrives, timer expires or the query is aborted. The transit agent state machine, as in the Update Protocol, is trivial.

The decision of whether to cache the message contents or not, and the decision of who to forward the message is done using the Query Message Forwarding Table (QMFT). There is one

QMFT for each of the four types of agents. The QMFT describes the actions to be taken on the receipt of a particular message from a particular agent.

Currently, the Query-Response protocol is used in map query, route request, locator request and adjacency request (see section 2). We note that the QMFT-based protocol specification approach provides flexibility by making it easy to change the functionality and the mapping - you just need to alter the entries in the table. The Q-R Protocol is typically used by Nimrod agents or other "users" in order to initiate queries. We use the term *client* to denote these users.

The Q-R protocol consists of one message - the QRMsg. The same format is used for both a Request Message (ReqMsg) used to request information, and as a Response Message (RspMsg) used to respond to the requested message. The QRMsg consists of a variable length header (NimQRHdr) and the database specific information (QRInfo). In this section, we only describe the QRHdr. QRInfo packet formats are specified in section 3.2.

7.1 Query/Response Header

Each item in the common update header is given below along with either the number of bits in the packet used by the item or a reference to a variable length packet format defined elsewhere.

```
NimQRHdr ::=
{
   org-agent : 8,
   dst-agent : 8,

   flags :16,
   db-type  : 8,

   count   : 8,
   opcode :16,

   org-eid :NimEID,
   org-loc :NimLoc,
   sign :NimSig
}
```

The packet field semantics are as follows.

- **org-agent.** The type of agent originating the query/response. Agent types are defined in [5].
- **dst-agent.** The type of agents for whom the query/response is intended.
- **flags.** Miscellaneous operation dependent flags.
- **db-type.** The type of the database that is being queried. At most one kind of database can be queried with a QRMsg. Database types are defined in [5].

- **count.** Operation dependent.
- **opcode.** In query, operation specific or database specific information; in response, zero if query can be responded to successfully, otherwise an error code indicating the reason for failure.
- **org-eid**. EID of the agent that issued the query/response. The format of NimEID is given in [5].
- **org-loc**. Locator of the agent that issued the query/response. The format of NimLoc is given in [5].
- **Sign**. Authentication field. Contains the signature of the agent originating the query/response. The format of NimSig is given in [5].

7.2 Agent Protocol Engines

A QRMsg is originated by a single agent (*originating agent*) and handled by a number of intermediate agents (*transit agent*) enroute to the destination(s). We give briefly the algorithm executed at each of the transit and originating agents.

7.2.1 Transit Agent

Given below is the algorithm executed by the originating agent (ThisAgent) upon receipt of Query Message (QryMsg) from another agent (PrevAgent). The QMFT is described in section 7.3.

```
begin

    /* Consult the QMFT table for caching and forwarding directions */

    NextAgentType = QMFT(ThisAgent, PrevAgent, QRMsg)

    if NextAgentType is not a valid agent, discard message.

    repeat
       choose a (previously unchosen) agent X of NextAgentType.
       if none found
       then trigger SNMP event and discard message
       else forward QryMsg to X.
   until successfully forwarded or message discarded.

end
```

7.2.2 Originating Agent Protocol Engine

The originating agent protocol engine has two states INITIAL and WAITING. Three interactions are used for interaction with the client :

1. DoQuery : A query from the client.
2. AbortQuery : An indication from the client to to abort a query. Even if the protocol manages to get a response, it is discarded.
3. QueryFail : A message from the protocol to the client indicating that a query could not be answered.

We note that each query will initiate a new state machine. We assume that messages are demultiplexed apriori to the "right" state machine.

```
begin

   receive process message

   state INITIAL :
      if it is DoQuery message from client
         Prepare QryMsg
         set query-timer with value from application or configuration
         Execute Transit Agent Protocol Engine (see section~\ref{TranAgen})
         Transition to state WAITING.
      if it is any other message
         Discard the message

   state WAITING :
       if it is RspMsg from peer
          deliver database contents to client;
          Transition to state INITIAL;
       if it is AbortQuery from client
          Transition to state INITIAL;
       if it is query-timer expiration
          send a QueryFail message to client;
          Transition to state INITIAL;
      if it is any other message
          Discard the message.
end.
```

7.3 The Query Message Forwarding Table (QMFT)

For each of four agent types (Forwarding Agent, Endpoint Rep., Node Rep. and Route Agent), we give below the actions upon receipt of an QRMsg. The agents are represented by F, Er, Nr and R and suffixed by p, c or s to indicate the agent in the parent, child or sibling node of the node under consideration. No suffix means the agent belongs to the node under consideration.

The QMFT is for queries only. All responses are sent directly to the querier and therefore do not need to invoke the Q-R protocol.

For each agent X, the QMFT consists of 4 columns, details of which are as follows.

1. **Database type**. The db-type in message received by X.

2. **PrevAgent**. The agent from which the message came at the level of THIS protocol. Note that the message may have hopped through many intermediate agents, but this is like a “tunnel” for this protocol. If the message is originated at this agent, then there is actually no PrevAgent, but we use PrevAgent = Client to denote this.

   Note that a boundary forwarding agent for a given node acts also as an interior forwarding agent for its parent node. An unsubscripted F always represents the *boundary* forwarding agent and F_c is an *interior* forwarding agent for the node in question. Sometimes, to emphasize the “interior-ness” or “boundary-ness” of a forwarding agent, we shall prefix by an ‘i’ or a ‘b’. The reader should understand that the following equivalences hold: bF = iF_p = F; iF = bF_c = F_c

3. **Actions**. This says what agent X will do when the message is received from PrevAgent. Sometimes uses functions “forward”, and “send” whose semantics are described following the table. whose semantics are indicated following the table. Combinations not given represent ”Unexpected” situations. If these happen, an SNMP event is triggered and the message is discarded.

4. **Fig,Ref**. Should be ignored for ascii (.txt) versions. For postscript(.ps) version, references a pictorial representation of the action. The figure number y is identified as F-y. In the figure, messages from one agent to another are represented by labelled arrows. In the QMFT, the relevant incoming arrows to X are denoted by Iarw <label1>, <label2> ... and the relevant outgoing arrows to X by Oarw <label1>, <label2> etc. Labels x and -x denote mutually exclusive actions (e.g., Oarw -x is done if Oarw x is not possible).

```
                         QMFT for F
                       --------------
                       **************
==========================================================================
  DB Type    PrevAgent             Actions                   Fig, Ref
==========================================================================
   Map       R(Querier)                                  F-2 Iarw 1
                             forward (QRMsg, Nrp})...........Oarw 2
--------------------------------------------------------------------------
 Locator        Nr                                        F-4 Iarw 1
                             forward (QRMsg, Fs)..............Oarw 2
--------------------------------------------------------------------------
 Locator        Fs            extract scope S from QRMsg  F-4 Iarw 2
                              send (QRMsg, S).................Oarw 3
--------------------------------------------------------------------------
 Route       Client           make QRMsg from RtReq       F-3
(RtReq)                       send (QRMsg, R(scope))..........Oarw 1
--------------------------------------------------------------------------


                          QMFT for Nr
                          -----------
                          ***********
```

```
==========================================================================
  DB Type     PrevAgent           Actions                       Fig Ref
==========================================================================
   Map        F                                               F-2 Iarw 2
                               Make QRMsg (Response)
                               send (QRMsg, R(Querier))....... F-2 Oarw 3
--------------------------------------------------------------------------
 Locator      Client           make QRMsg using LocReq         F-4
                               forward (QRMsg, F)..................Oarw 1
--------------------------------------------------------------------------
 Locator        F              make QRMsg (response)           F-4 Iarw 3
                               send (QRMsg, Querier)...............Oarw 4
-------------------------------------------------------------------------
 Locator      Er(querier)      make QRMsg (response)           F-4 Iarw 5
                               send (QRMsg, Querier)...............Oarw 6
-------------------------------------------------------------------------


                                QMFT for Er
                                ----------
                                **********
==========================================================================
  DB Type     PrevAgent               Actions                       Fig Ref
==========================================================================
 Locator      Client            make QRMsg using Loc. Req.       F-4
                                send (QRMsg, Nr(scope)).............Oarw 5
-------------------------------------------------------------------------


                                  QMFT for R
                                  -----------
                                  ***********
===========================================================================
  DB Type     PrevAgent             Actions                         Fig,Ref
===========================================================================
  Map         Application       Z=Node whose map is reqrd      F-2
(MapRqst)                       make QRMsg using MapRequest
                                send (QRMsg, Z)..................Oarw 1
--------------------------------------------------------------------------
 Route        F (Querier)       generate or find cached route F-3 Iarw 1
                                 make QRMsg (response)
                                 send (QRMsg, querier)...........Oarw 2
-------------------------------------------------------------------------

Semantics of functions:

  forward (Msg, AgentSet) :
        forward Msg *reliably* to one agent of each type given in Agent Set.
```

```
      Includes the functionality of resending to another agent if one
      agent cannot be reached. The set {all X} indicates all agents of
      type X including itself. May involve multiple hops. If an agent A
      in AgentSet is not 1 hop away, Msg is handed over to a neighboring
      forwarding agent that knows how to reach A.

  send (Msg, Node) :
      conveys Msg to a boundary forwarding agent in Node. May need to
      set up a path to do this.
```

8 Discovery Protocol

To Be Defined.

9 APPENDICES

9.1 Appendix 1 : Figures for Update and Q-R protocols

Figure 1 : Map Update

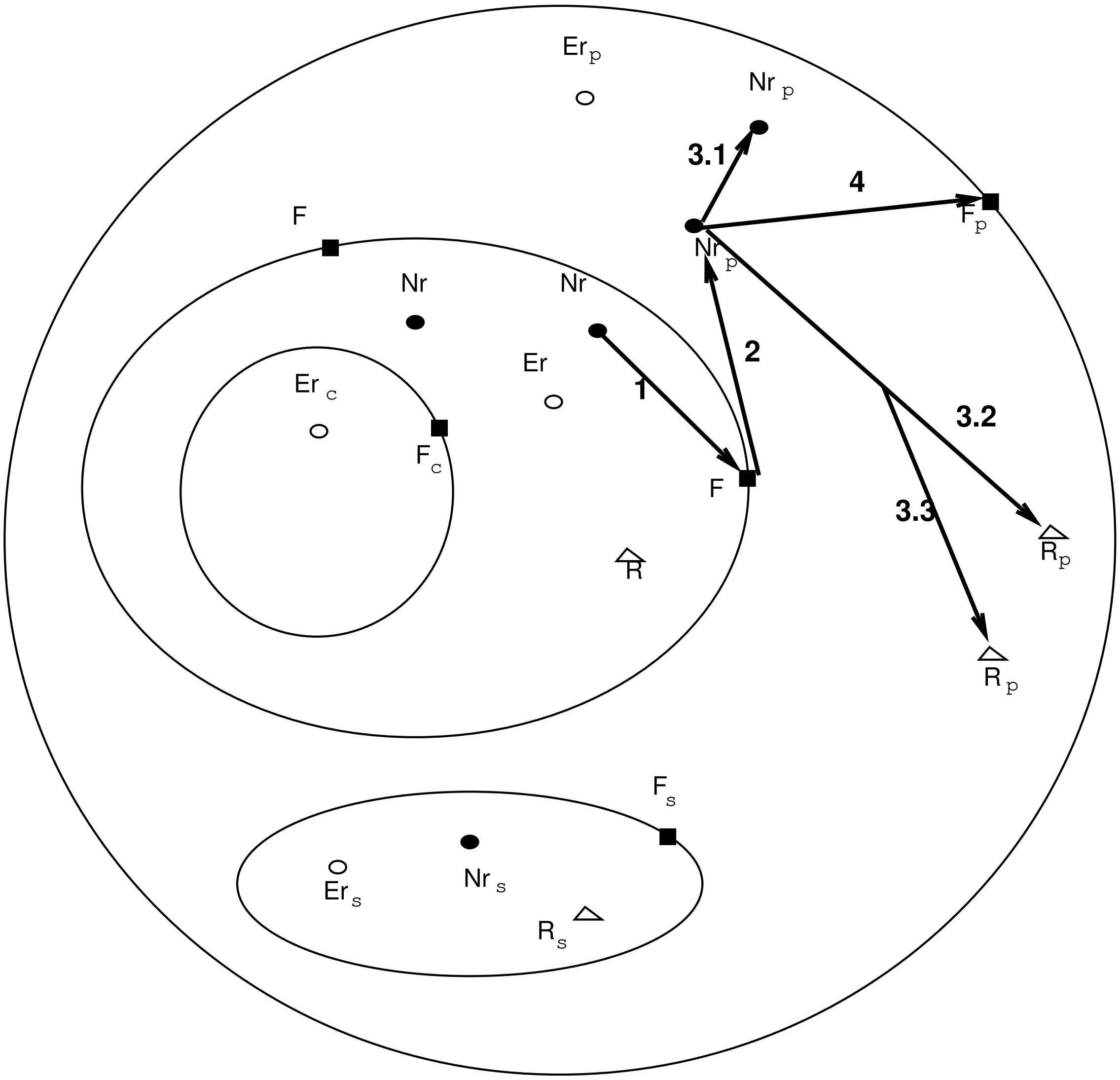


Figure 2 : Map Query

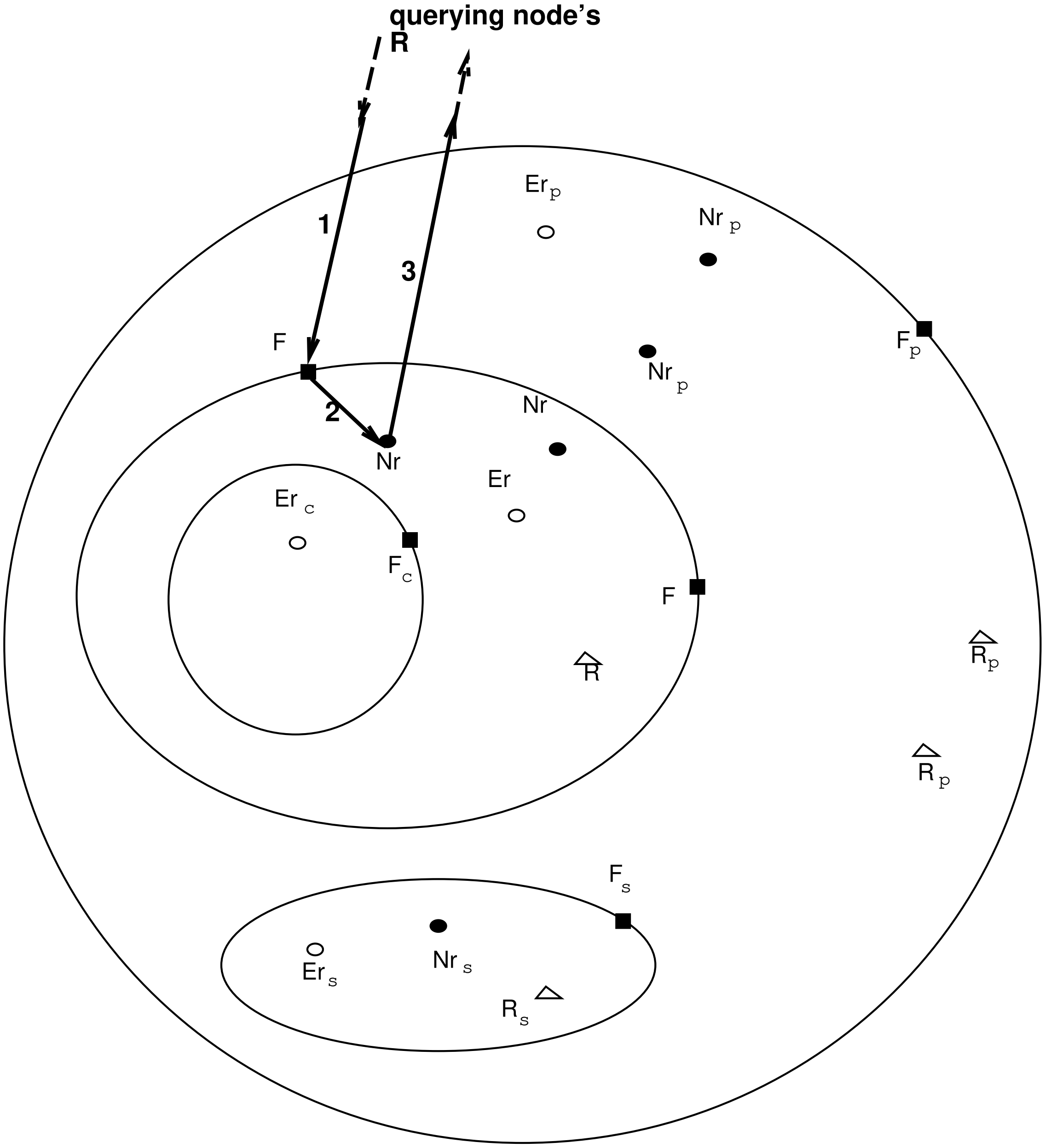


Figure 3 : Route Request

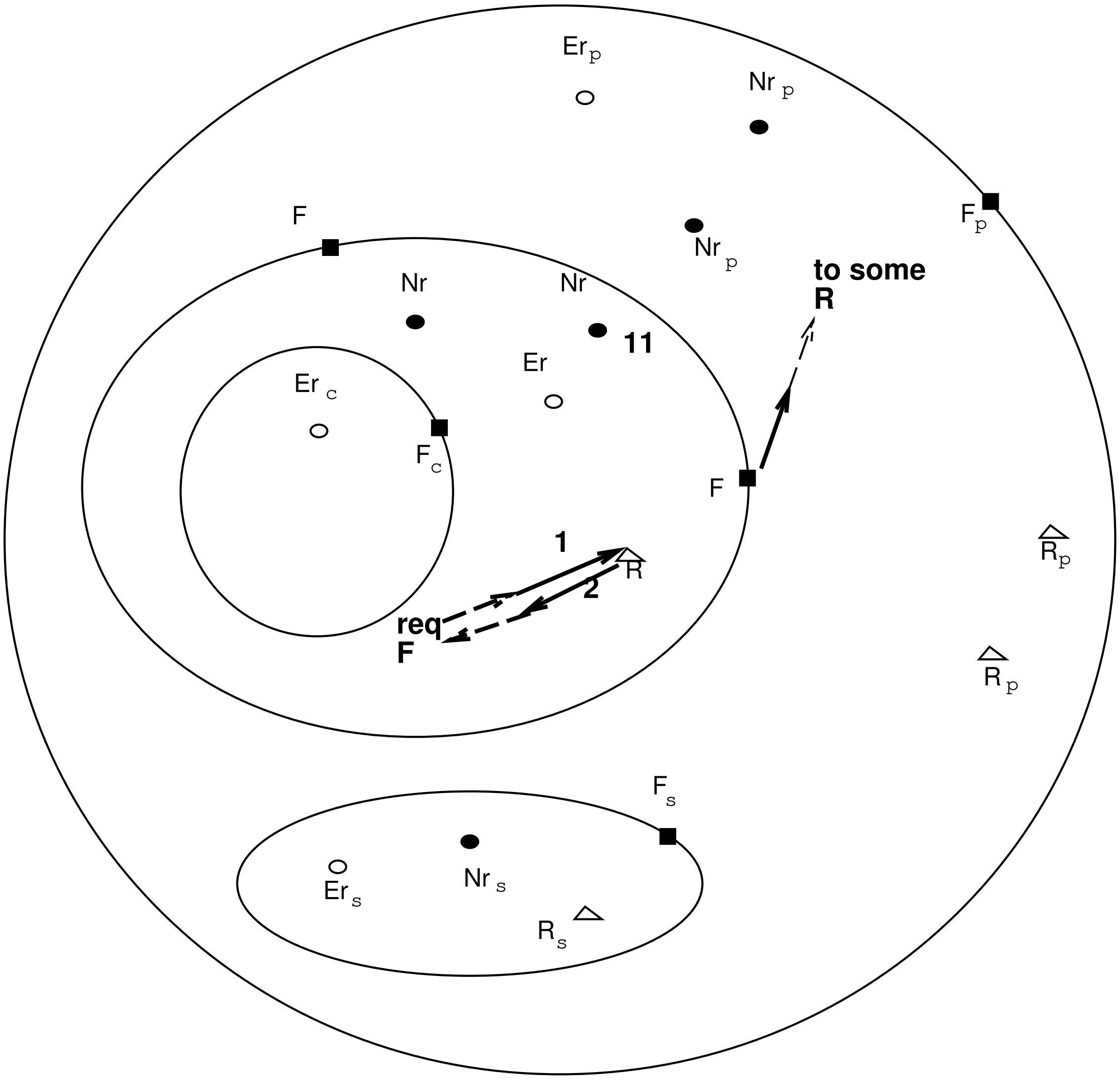


Figure 4 : Locator Request (Ph 1 − Acquisition)
Arcs 1−4 denote Node Locator Acq, 5−6 denote EP Locactor Acq
(note : assumes locator req. scope is the outermost node below)

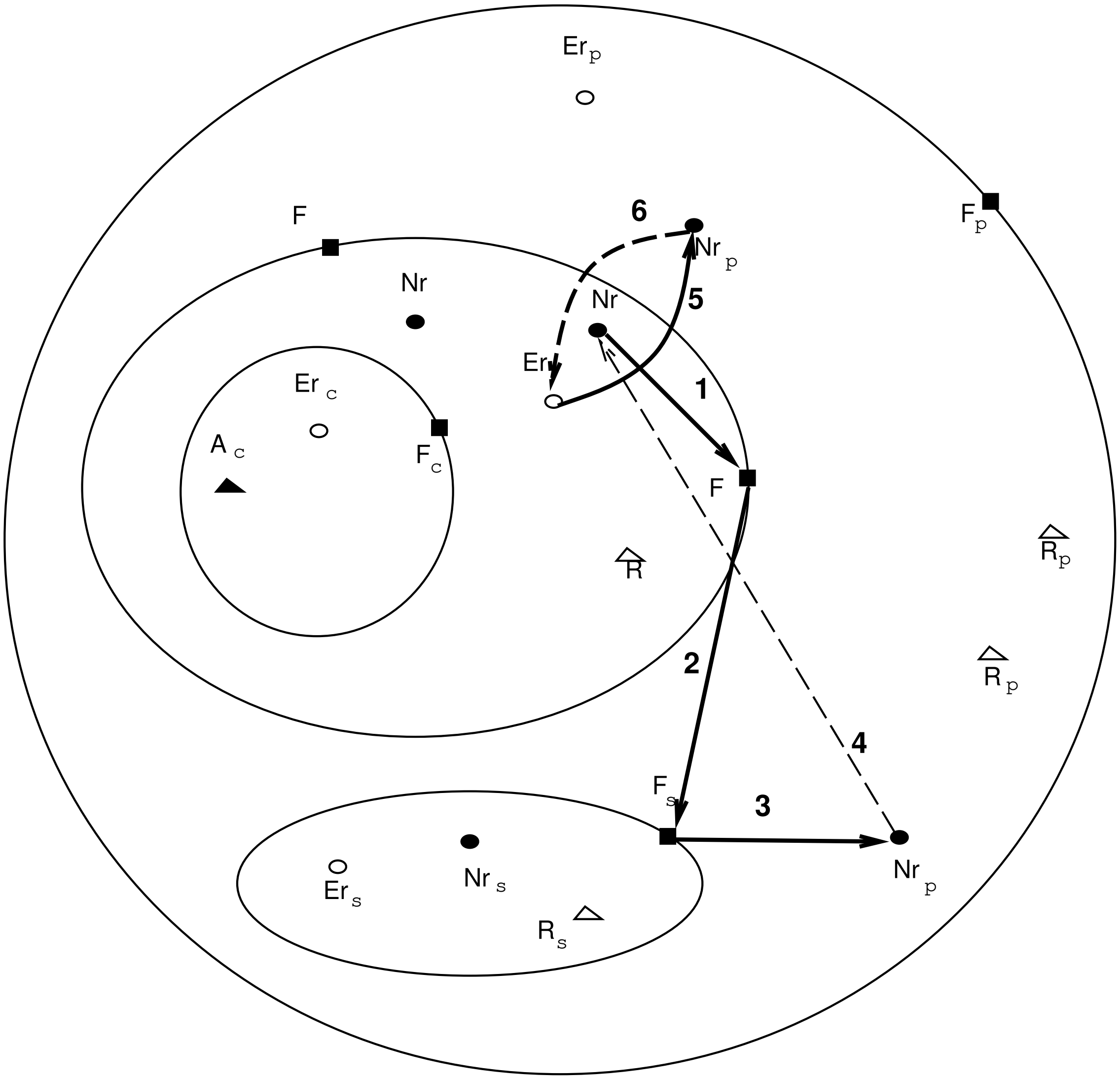


Figure 5 : Locator Request (Ph 2 : Notification)

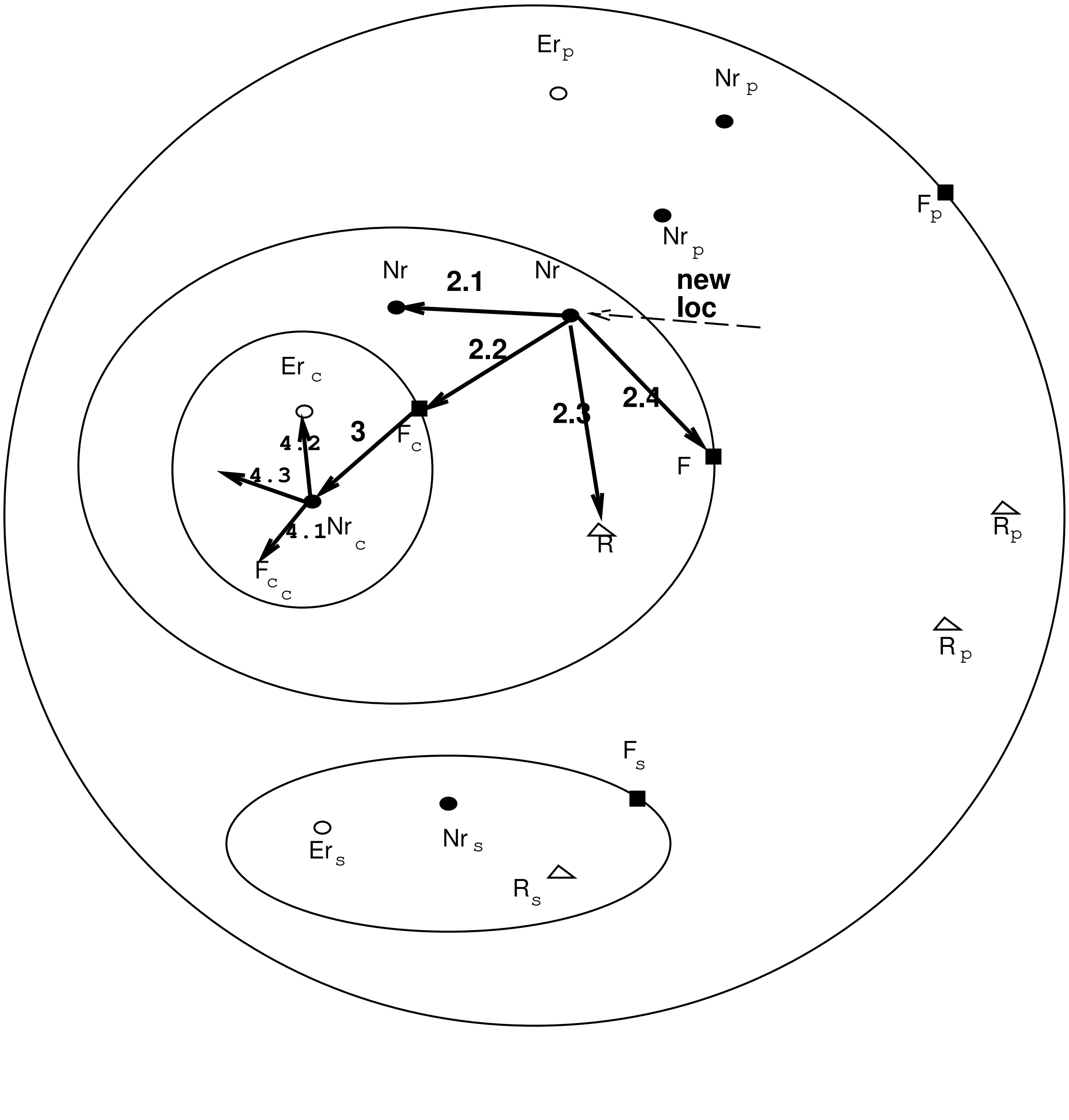
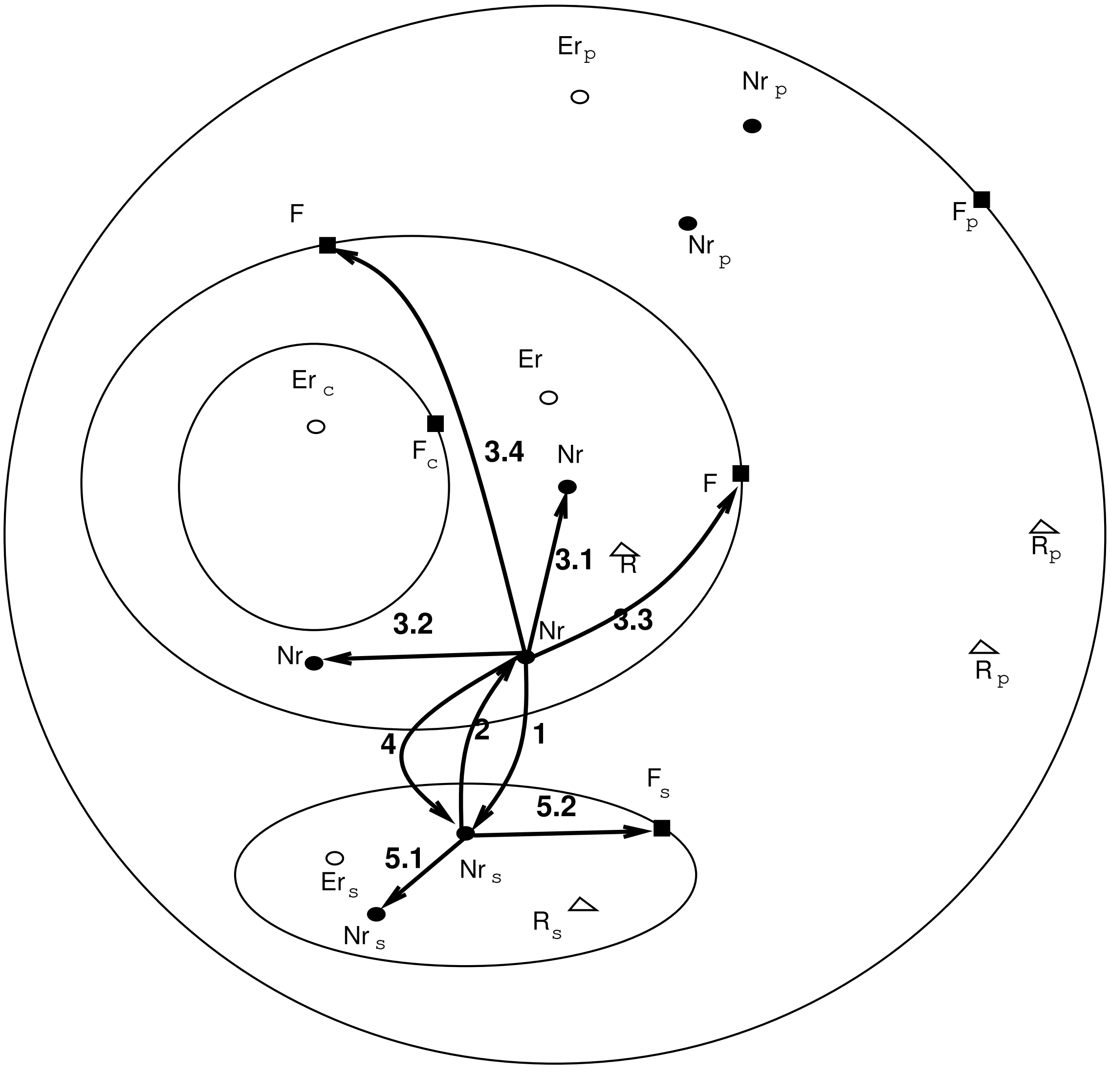


Figure 6 : Adjacency Formation



9.2 Appendix 2 : TTCP Example

The
following code is excerpted from a test program by Braden and Wei:

```
/*
 *  Client test program for T/TCP.
 *
 *      Bob Braden & Liming Wei,
 *      USC Information Sciences Institute, 1992, 1993, 1994
 *      Work supported by NSF.
 *
 */


transact(requestbuf, requestlen, peeraddr_in,
rcvbuf, rcvlen, myaddr_in)
char *requestbuf, *rcvbuf;
int  requestlen, rcvlen;
sockaddr_in *myaddr_in, peeraddr_in;
{

   if (0==(so = getsocket(socktype[SOCK_STREAM], myaddr_in))) {
           printf("Error termination\n\n");
           return;
   }
   tolen = sizeof (struct sockaddr_in);

   /* T/TCP: send entire request in one sendto call
    * (not necessary) and set EOF true on (last) call.
    * Note that this leaves TCP connection half open,
    * and that the EOF can be used by the server to
    * delimit the request message.
    */
    error = sendto(so, requestbuf, requestlen, MSG_EOF,
                     peeraddr_in, tolen);
    if (error < 0) {
        perror("sendto");
        exit(1);
    }


    i = read(so, rcvbuf, rcvlen));

    if (i < 0){
         perror("read");
         exit(1);
```

```
    }

   close(so);
}

getsocket(type, addr_inp)
        int type;
        struct sockaddr_in *addr_inp;
        {
        int so, one = 1;
        int addrlen = sizeof(struct sockaddr_in);

        if (0>(so = socket(AF_INET, type, 0))){
                perror("socket");
                exit(2);
        }
        addr_inp->sin_family = AF_INET;
        addr_inp->sin_addr.s_addr = INADDR_ANY;

        if (0 > setsockopt(so, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(int))) {
                perror("SO_REUSE");
                exit(2);
        }

        if (0 > bind(so, addr_inp, sizeof(struct sockaddr_in))){
                perror("bind error");
                if (errno == EADDRINUSE) {
                        close(so);
                        return(0);
                }
                exit(4);
        };
        return(so);
}
```

9.3 Appendix 3 : An XDR definition of Map Database

9.3.1 An XDR Definition of the Map Database

The external representation of the map database defined here should make it possible to interoperate implementations that do not understand the same set of service and restriction specifications. An implementation should be able to recognize what elements are understood, and in the case that a given element is not understood should handle it in a reasonable manner: for example, it should be able to tell if this element should be forwarded or dropped.

This section specifies the format for the basic routing entities: maps, nodes, and connectivity specifications.

```
/* the elements appear top-down */
/* note that variable sized arrays can have zero elements */




/* Nodes */
struct node
{
   locator id;
/* the locator of the node */

   locator adjacentSources<>;
/* a variable sized array of neighbor nodes' locators that can send
   data to this node */

   locator adjacentDestination<>;
/* a variable sized array of neighbor nodes' locators to which this
   node can send data */

   ConSpec transitConSpecs<>;
/* the transit connectivity specifications; this item and the next two
   items are variable size array of connectivity specifications */

   ConSpec incomingConSpecs<>;
   ConSpec outgoingConSpecs<>;

   int numberMaps;
/* the number of maps a node has */

   int internalMapIDs<>;
/* a variable sized array of integers that contains a list of map ids
   */
}
```

```
/* Maps */

struct map
{
   int number;
/* the maps identifiying number */

   node mapNodes<>;
/* a variable length array of nodes */
}



/* Connectivity Specifications */
struct conSpec
{
   locator id;
/* locator of the connectivity specification */

   locatorSet inputPrefixes<>;
/* indicates the set of source locators from which this connectivity
   Spec can be used */

   locatorSet outputPrefixes<>;
/* indicates the set of locators to which this connectivity Spec can
   be used */

   performanceSpec performance<>;
/* variable sized array of performance specifications, e.g. average
   100 ms delay */

   restrictionSpec restriction<>;
/* variable sized array of restriction specifications, e.g., only
   research traffic */
}



/* Performance Specifications */

/* The format for performance and restriction specifications is designed to
   enable graceful evolution by facilitating the gradual introduction
   of new types of performance and restriction specifications. This is done
   by giving to performance and restriction specifications a common initial
   portion consisting of two fields: a handling code and a length. The
   handling code advises about what to do with specifications that an
```

```
   implementation does not yet understand. The length field is
   required to find the extend of unknown specifications.
*/


/* The example below assumes an implementation that knows of three
   different kinds of performance specifications and two different
   kinds of restriction specifications
*/


/* Performance Specifications */

/* An enumeration type to name the known performance specifications */

enum perfSpecKind
{
   BESTEFFORT = 1,
   DELAY = 2,
   BANDWIDTH = 3
};


union performanceSpec switch (perfSpecKind kind)
{
 case BESTEFFORT:
   bestEffortSpec spec;

 case DELAY:
   delaySpec spec;

 case BANDWIDTH:
   bandwidthSpec spec;

 default:
   unknownPerformance performance;
}

struct bestEffortSpec
{
   handlingCode code;

   int length;

   void;
/* no extra information */
}
```

```
struct delaySpec
{
   handlingCode code;

   int length;

   int delay;
/* delay measured in ms. */
}

struct bandwidthSpec
{
   handlingCode code;

   int length;

   int bandwidth;
/* bandwidth in kb/s */
}

struct unknownPerformance
{
   handlingCode code;

   opaque contents<>;
/* notice that this contains a length field */
}


/* Restriction Specifications */

/* An enumeration type to name the known restriction  specifications */

enum restrictionSpecKind
{
   UNRESTRICTED = 1,
   RESEARCH = 2
};

union restrictionSpec switch (restrictionSpecKind kind)
{
case UNRESTRICTED:
   unrestrictedRestriction restriction;

case RESEARCH:
   researchRestriction restriction;

default:
```

```
   unknownRestriction restriction;
}

struct unrestrictedRestriction
{
   handlingCode code;

   int length;

   void;
/* no further information */
}

struct researchRestriction
{
   handlingCode code;

   int length;

   void;
/* no further information */
}

struct unknownRestriction
{
   handlingCode code;

   opaque contents<>;
}


/* Basic Elements */

typedef opaque locator<>;
/* the representation of locators if TBD */

enum setKind
{
   ALL = 1,
   INCLUDELIST = 2,
   EXCLUDELIST = 3,
};

union locatorSet switch (setKind kind)
{
 case ALL:
   void;
/* no further information is required the meaning of ``all'' is
```

```
   context dependent */

 case INCLUDELIST:
   locator list<>;
/* variable sized array of locators containing a explicit list of
   those locators to be included */

 case EXCLUDELIST:
   locator list<>;
/* variable sized array containing a explicit list of those locators
   to be excluded */
}

typedef opaque handlingCode[4];
/* the representation of the handling codes is TBD */
```

9.3.2 Queries

Queries are used to obtain map information from node representatives. The query format defined below identify specific data items, and combinations or subsets of data items. Queries are specified by given an XDR declaration of the query itself and of the associated reply. The queries defined below cannot express requests that depend on the values of the elements. For example, request of the style “send all connectivity specifications with property foo” are not supported.

```
enum queryKind
{
   NODE = 1,
   TOPOLOGY = 2,
   MAP = 3
};

union query switch(queryKind kind)
{
 case NODE:
   void;
/* no further information required */

 case MAP:
   integer mapID;
/* the id of the requested map */

 case TOPOLOGY:
   integer mapID;
/* the id of the requested map */
}
```

```
/* for each query the following is a specification of the expected
answer */

union reply switch(queryKind kind)
{
 case NODE:
   node thisNode;

 case MAP:
   map thisMap;

 case TOPOLOGY:
   topology thisTopology;
}

struct topology
{
   nodeSkeleton sNode<>;
/* a variable sized array of nodeSkeleton */
}

struct nodeSkeleton
{
   locator id;
/* the locator of the node */

   locator externalSources<>;
/* a variable sized array of locators of external neighbor nodes that
   can send data to this node */

   locator adjacentDestination<>;
/* a variable sized array of neighbor nodes to which this node can
   send data */
}
```

References

[1] I. Castineyra, J.N. Chiappa and M. Steenstrup, “The Nimrod Architecture”, Working Draft, Mar. 1995.

[2] M. Steenstrup, “A Perspective on Nimrod Functionality”

[3] S. Ramanathan, “Mobility Support for Nimrod”, Working-Draft, March 1995.

[4] S. Ramanathan, “Multicast Support for Nimrod”, Working-Draft, March 1995.

[5] C. Lynn, "Nimrod Module Interface Specifications"