Chapter 1

LABEL Forwarding

1.1 Introduction

In this report, we demonstrate the implementation of LABEL forwarding algorithm in Haggle framework.

1.2 LABEL forwarding strategy

In real world, people from the same affiliation/community tends to meet more frequently than people outside the affiliation/community. The LABEL forwarding strategy has been proposed to find a good forwarders to relay messages to the other members in the same affiliation/community.

The LABEL is a simple forwarding strategy. Here we assume that each node has a LABEL, which is used to indicate its community. The internal algorithm always forwards messages to destinations, or to next-hop nodes belonging to the same label as the destinations (as shown in Algorithm 1) .

```
for each_EncounteredNode do
    if LabelOf(EncounteredNode) == LabelOf(destination) then
        SetAsDelegates(EncounteredNode);
    end
end
```

Algorithm 1: How to write algorithms

The advantage of LABEL forwarding strategy is that it requires very little routing information and is easy to be implemented.

1.3 Implementation

Haggle provides a framework to run Delay and Tolerant Networks (DTNs) or Opportunistic Networks. Haggle framework has been chosen as the base architecture for our LABEL forwarding strategy. Our LABEL implementation has been developed specifically for the Linux testbed. The source code has been released via the Google code repository in http://code.google.com/p/haggle-label/.

The key implementation relies in Haggle kernel, which is in charge of managing the forwarding, network, connectivity, contacts, neighbours and more. At the moment, the Haggle kernel only supports Prophet forwarding module[1] as default. We aimed at replacing the Prophet forwarding module with our LABEL module.

In line with the Haggle framework, the ForwarderLabel class is inherited from the ForwarderAsynchronous class, and implements some functions as indicated in the Forwarder class (see Figure 1.1).

The following functions are implemented in ForwarderLabel class:

- newRoutingInformation() parses metadata containing metrics received from neighbor.

```
bool ForwarderLabel::newRoutingInformation()
{

   if (this is new node)
      add node_id to id_to_string table;

   while (RoutingInformation) {

      RoutingInformation->getParameter(LABEL);

      save the node_id and LABEL parameter to routing table;

      go to next RoutingInformation;

   }

   return true;
}
```

Listing 1.1: newRoutingInformation()

- addRoutingInformation() adds new routing information that you want to give to neighbor.

```
bool ForwarderLabel::addRoutingInformation()
{

   Add node_id to RoutingInformation;

   Add myLabel to RoutingInformation;

   return true;

}
```

Listing 1.2: addRoutingInformation()

- _generateDelegatesFor() generates a list of neighbors that are good forwarders for the target. The other_targets parameter is a list of already found targets. It returns delegates in an event.

```
void ForwarderLabel::_generateDelegatesFor()
{
```

[1]However, there is currently no dynamic module loading in Haggle. ForwarderLabel module is made default by replacing the ForwarderProphet.

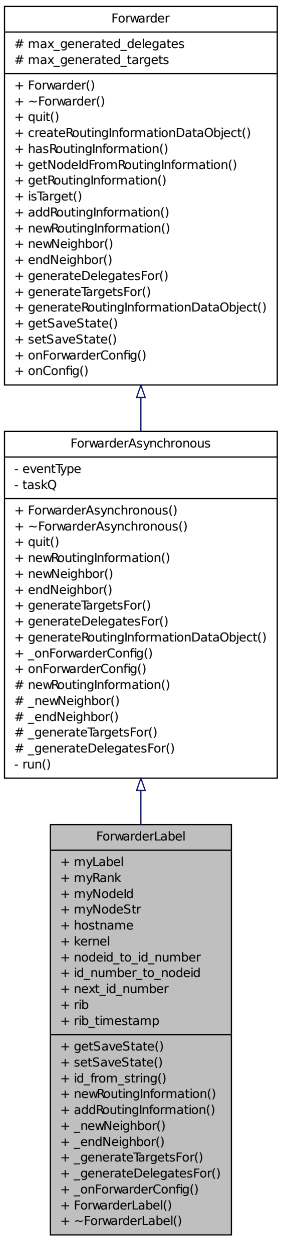

Figure 1.1: Inheritance diagram for ForwarderLabel

```

    create sorted_delegate_list;

    get the target_id;

    get the target_label;

    for (each node in routing table)
    {
       if (node is not target  && node is not current_node ) {

          if (neighborLabel==targetLabel)==0)
             {
                insert node into sorted_delegate_list;
             }
          }
       }
    }

    sort(sorted_delegate_list);

    kernel->addEvent(EVENT_TYPE_DELEGATE_NODES);

}
```

Listing 1.3: generateDelegatesFor()

- _generateTargetsFor() generates a list of targets that a neighbor is a good delegate for. Return targets in an event.

In LABEL the routing table is a mapping between node_id and LABEL. Each node will maintain a table like this. As long as the nodes exchange the routing information, It will send it LABEL to each other. When it receiving new routing information, the nodes will add a new node_id and LABEL to its new routing table. The implementation of LABEL algorithm has been built on delegation forwarding. Finally the nodes which are created as quality of the node as relay are generated as the delegates.

Chapter 2

ForwarderLabel Class

2.1 ForwarderLabel Class Attributes

A number of data structures are defined in ForwarderLabel Class.

- LABEL_T myLabel
- RANK_T myRank
- bubble_node_id_t myNodeId
- string myNodeStr
- char hostname [HOSTNAME_LEN]
- HaggleKernel $*$ kernel
- Map$<$ string, bubble_node_id_t $>$ nodeid_to_id_number
- Map$<$ bubble_node_id_t, string $>$ id_number_to_nodeid
- bubble_node_id_t next_id_number
- bubble_rib_t rib
- Timeval rib_timestamp

2.1.1 char ForwarderLabel::hostname[HOSTNAME_LEN]

The hostname stores the hostname for this node.

2.1.2 HaggleKernel$*$ ForwarderLabel::kernel

The kernel is a haggle kernel structure.

2.1.3 LABEL_T ForwarderLabel::myLabel

The myLabel stores the LABEL information for this node.

2.1.4 bubble_node_id_t ForwarderLabel::myNodeId

The myNodeId stores the NodeId for this node.

2.1.5 string ForwarderLabel::myNodeStr

The myNodeStr stores the StringId for this node.

2.1.6 bubble_node_id_t ForwarderLabel::next_id_number

The next_id_number is the id number which is free to be used.

2.1.7 Map<string, bubble_node_id_t> ForwarderLabel::nodeid_-to_id_number

The nodeid_to_id_number is a mapping structure used to convert from the NodeStringId to the NodeId

2.1.8 bubble_rib_t ForwarderLabel::rib

The rib is the local forwarding metrics table.

2.1.9 Map<bubble_node_id_t, string> ForwarderLabel::id_-number_to_nodeid

The id_number_to_nodeid is a mapping structure used to convert from the NodeId to the NodeStringId

2.1.10 Timeval ForwarderLabel::rib_timestamp

The rib_timestamp stores the time information

2.2 ForwarderLabel Class Member Functions

ForwarderLabel class inherits a number of member functions from the super class.

- size_t getSaveState (RepositoryEntryList &rel)
- bool setSaveState (RepositoryEntryRef &e)
- bubble_node_id_t id_from_string (const string &nodeid)
- bool newRoutingInformation (const Metadata ∗m)
- bool addRoutingInformation (DataObjectRef &dObj, Metadata ∗parent)
- void _newNeighbor (const NodeRef &neighbor)
- void _endNeighbor (const NodeRef &neighbor)
- void _generateTargetsFor (const NodeRef &neighbor)

- void _generateDelegatesFor (const DataObjectRef &dObj, const NodeRef &target, const NodeRefList ∗other_targets)
- void _onForwarderConfig (const Metadata &m)
- ForwarderLabel (ForwardingManager ∗m=NULL, const EventType type=-1)
- ∼ForwarderLabel ()

2.2.1 Constructor and Destructor Functions

Constructor is called every time you create ForwarderLabel module, and destructor is called every time you destroy ForwarderLabel module.

2.2.2 ForwarderLabel::ForwarderLabel (ForwardingManager ∗ *m* = `NULL`, const EventType *type* = `-1`)

2.2.3 ForwarderLabel::∼ForwarderLabel ( )

2.2.4 Class Member Function

2.2.5 void ForwarderLabel::_newNeighbor (const NodeRef & *neighbor*) `[virtual]`

The _newNeighbor is called when a neighbor node is discovered.

2.2.6 void ForwarderLabel::_endNeighbor (const NodeRef & *neighbor*) `[virtual]`

The _endNeighbor called when a node just ended being a neighbor.

2.2.7 void ForwarderLabel::_generateDelegatesFor (const DataObjectRef & *dObj,* const NodeRef & *target,* const NodeRefList ∗ *other_targets*) `[virtual]`

The _generateDelegatesFor function generates an EVENT_TYPE_DELEGATE_NODES event to provide all the nodes that are good delegate forwarders for the given node.

This function is given a target to which to send a data object, and answers the question: To which delegate forwarders can I send the given data object, so that it will reach the given target?

If no nodes are found, no event should be created.

2.2.8 void ForwarderLabel::_generateTargetsFor (const NodeRef & *neighbor*) `[virtual]`

The _generateTargetsFor function generates an EVENT_TYPE_TARGET_NODES event to provide all the target nodes that the given node is a good delegate forwarder for.

This function is given a current neighbor, and answers the question: For which nodes is the given node a good delegate forwarder?

If no nodes are found, no event should be created.

2.2.9 void ForwarderLabel::_onForwarderConfig (const Metadata & *m*) `[virtual]`

The _onForwarderConfig function reads the configuration from config.xml file.

2.2.10 bool ForwarderLabel::addRoutingInformation (DataObjectRef & *dObj,* Metadata $*$ *parent*) `[virtual]`

The addRoutingInformation function is used to generate the Metadata containing routing information which is specific for that forwarding module.

2.2.11 bool ForwarderLabel::newRoutingInformation (const Metadata $*$ *m*) `[virtual]`

The newRoutingInformation function is used when a data object has come in that has a "Routing" attribute.

Also called for each such data object that is in the data store on startup.

Since the format of the data in such a data object is unknown to the forwarding manager, it is up to the forwarder to make sure the data is in the correct format.

Also, the given metric data object may have been sent before, due to limitations in the forwarding manager.

2.2.12 size_t ForwarderLabel::getSaveState (RepositoryEntryList & *rel*) `[virtual]`

The getSaveState function gets saved forwarding metrics table

2.2.13 bool ForwarderLabel::setSaveState (RepositoryEntryRef & *e*) `[virtual]`

getSaveState function saves forwarding metrics table

2.2.14 bubble_node_id_t ForwarderLabel::id_from_string (const string & *nodeid*)

The id_from_string function get the bubble_node_id_t from the NodeStringId. If the bubble_node_id_t wasn't in the map to begin with, it is inserted, along with a new id number.

Index