Introduction to the geographic greedy forwarding

What is the geographic greedy forwarding?

There are several types of ad hoc routing protocols, such as proactive, reactive, geographic stateless routing, and so on. I believe geographic stateless routing is the simplest routing protocol to be implemented in ns2. Why it is the simplest? It is because the property of stateless indicates that no node in a network does not maintain any routing table. Note that we will still keep neighbors list at the routing component, but such information is available at logical link layer.

The purpose of this series is to learn how to implement a new routing protocol, thus we consider static networks where nodes are not move. The assumptions we hold for implementation include; 1) all nodes have its own geographic location (2-dimensional coordinates); 2) each node knows its neighbors location; and 3) a source node knows the ID and the location of the corresponding destination.

The geographic greedy forwarding protocol that we are going to implement works as follow.

1. On receiving a packet from upper layer, the source node adds a header including the destination ID and location.
2. The source node sends out the packet to the neighbor closest to the destination.
3. On receiving a packet from a neighbor, a node forwards the packet if there exists a neighbor closer to the destination than itself.
4. If there is no neighbor closer than itself, so called local minimum, it simply drops the packet.
5. When the packet arrives at the destination, routing is done.

Source code

In this project, we will write new codes (greedy_pkt.h, greedy.h, greedy.cc, greedy_nbr.h greedy_nbr.cc, greedy_flow.h, and greedy_flow.cc), new script codes (greedy.tcl (the scenario file) and nbr.tcl), and modify existing source codes (common/packet.h, queue/priqueue.cc, tcl/lib/ns-lib.tcl, tcl/lib /ns-packet.tcl, and tcl/lib/ns-mobilenode.tcl).

The source code of this project is greedy.tar.gz.

Packet format

In a packet, we need to define the source node ID, the destination node ID, and the location of the destination. The following is the part of greedy/greedy_pkt.h.

```
struct hdr_greedy_data {
    GreedyPktType type_;                // the type of pkt
    nsaddr_t src_;                      // the src addr
    nsaddr_t dest_;                     // the dest addr
    float destX_;                       // the location X of the dest
    float destY_;                       // the location Y of the dest

    inline int size() {
        unsigned int s = 2 * sizeof(int) + 1 + 2 * sizeof(float);
        return s;
    }
};
```

Neighbor list and flow information

Since we assume that each node has its neighbors list including their location and the destination location, the routing module will need to have those information. Hence, we create two classes as follow.

```
#include "packet.h"

class GreedyNbr {
public:
    nsaddr_t addr_;     // The address
    float x_;           // x
    float y_;           // y

    GreedyNbr(nsaddr_t, float, float);
};
```

```
#include "config.h"

/*
 * This is the flow class for source nodes to learn destination location.
 * This is because that geographical routing protocols assume
 * a source node knows the location of its destination.
 */
class GreedyFlow {
public:
    nsaddr_t dest_;     // dest addr
    float destX_;       // dest location
    float destY_;

    GreedyFlow(nsaddr_t, float, float);
};
```

Routing Module

The routing module is the most important part of this project, because we are implementing a routing protocol. First, we need to make packet class to bind TCL and C++ as follow. You can consider this is a template, and when you implement another protocol, you just change the name of the class.

```
int hdr_greedy::offset_;

static class GreedyHeaderClass : public PacketHeaderClass{
public:
    GreedyHeaderClass() : PacketHeaderClass("PacketHeader/Greedy",
                                            sizeof(hdr_all_greedy)){
                                            bind_offset(&hdr_greedy::offset_);

    }
} class_greedyhdr;
```

Second, we need to bind the routing module in C++ and TCL so that you can create a routing agent from a scenario file. Again, you can consider this is a template. You will always need to do this when you implement a new routing protocol.

```
static class GreedyAgentClass : public TclClass {
public:
    GreedyAgentClass() : TclClass("Agent/Greedy") {}
    TclObject *create(int argc, const char*const* argv) {
        return (new GreedyAgent());
    }
} class_GreedyAgent;

GreedyAgent::GreedyAgent() : Agent(PT_GREEDY), addr_(-1) , x_(0.0), y_(0.0) {

}
```

Thrid, we need to override Agent::command(arg ....) so that we can access this class in C+ from a TCL file. For example, when you want to set up neighbor list from a TCL file, you can wirte "$node_($i) set-nbr $nbr_id $nbr_x $nbr_y". Note that the number of arguments is at least two, the first one is node instance, $node_($i), and the second one is the command such as "start", "set-location", and so forth. Then, the parameters that you want to pass to C++ code follow.

```
int GreedyAgent::command(int argc, const char*const* argv) {
    if (argc == 2) {
        if (strcasecmp(argv[1], "start") == 0) {
            return TCL_OK;
        } else if (strcasecmp(argv[1], "set-location") == 0){
            setLocation();
            return TCL_OK;
        }
    } else if (argc == 3) {
        if (strcmp(argv[1], "port-dmux") == 0) {
            port_dmux_ = (PortClassifier*)TclObject::lookup(argv[2]);
            if (port_dmux_ == 0) {
                fprintf(stderr, "%s: %s lookup of %s failed \n", __FILE__, argv
                return TCL_ERROR;
            }
            return TCL_OK;
        } else if (strcmp(argv[1], "node") == 0) {
            node_ = (MobileNode*)TclObject::lookup(argv[2]);
            if (node_ == 0) {
                fprintf(stderr,  "%s: %s lookup of %s failed \n", __FILE__, arg
                return TCL_ERROR;
            }
            return TCL_OK;
        } else if (strcmp(argv[1], "addr") == 0) {
            addr_ = Address::instance().str2addr(argv[2]);
            return TCL_OK;
        } else if (strcmp(argv[1], "log-target") == 0 || strcmp(argv[1], "trace
            logtarget_ = (Trace*)TclObject::lookup(argv[2]);
            if (logtarget_ == 0) {
                fprintf(stderr, "%s: %s lookup of %s failed \n", __FILE__, argv
                return TCL_ERROR;
            }
            return TCL_OK;
        }
    } else if (argc == 5) {
        if (strcmp(argv[1], "set-nbr") == 0) {
            addGreedyNbr(Address::instance().str2addr(argv[2]), atof(argv[3]),
            return TCL_OK;
        }
        if (strcmp(argv[1], "set-flow-data") == 0) {
            addFlow(Address::instance().str2addr(argv[2]) , atof(argv[3]), atof
            return TCL_OK;
        }
    }
    return Agent::command(argc, argv);
}
```

Fourth, we need to override Agent::recv(args ...). This is called when upper layer or lower layer pass a packet to the routing module. For example, when a CBR agent sends a packet, a routing agent receive the packet by this function. Also, when a node receive a packet from another node, the routing agent receive the packet from a link layer agent by this function.

In the first "if" statement, the agent checks if the source IP address is its own address. If so, the packet is either one that loops or one from the upper layer. If the number of forwarding in hdr_cmn (it stands for common header) is equal to 0, the packet is from upper layer, and thus the routing module set greedy_pkt header to be the corresponding destination information.

If the first "if" statement is is not true, the packet is definitely an incoming packet from another node. So, the routing module checks what kinds of packet it is. The type of a packet is stored at "hdr_cmn ch->ptype", which could be an application packet e.g., PT_CBR, a control packet i.e., PT_GREEDY, and so on. But, be careful. Greedy forwarding is stateless and therefore it does not introduce any control packet in the routing layer. So, the packet type could not be PT_GREEDY. If the packet is one generate by an application such as CBR, the routing agent forward the packet to the destination.

```
void GreedyAgent::recv(Packet* p, Handler* h) {
    struct hdr_cmn* ch = HDR_CMN(p);
    struct hdr_ip* ih = HDR_IP(p);

    if (ih->saddr() == addr_) {
        // if there exists a loop and the packet is not DATA packet, drops the
        if (ch->num_forwards() > 0 && ch->ptype() == PT_GREEDY) {
            // loop
            drop(p, DROP_RTR_ROUTE_LOOP);
            return;
        } else if (ch->num_forwards() == 0) {
            struct hdr_greedy_data *hdr = HDR_GREEDY_DATA(p);

            // else if this is a packet this node is originating
            // in reality, geographical routing does not use IP, so we do not
            ch->size() += hdr->size();

            GreedyFlow *td = getFlow(ih->daddr());

            hdr->type_ = GREEDY_PKT_DATA;
            hdr->src_ = addr_;
            hdr->dest_ = ih->daddr();
            hdr->destX_ = td->destX_;
            hdr->destY_ = td->destY_;

            ih->ttl_ = 100; // change ttl, the default ttl is 32
        }
    }
```

```
    // handling incoming packet
    if (ch->ptype() == PT_GREEDY) {
        struct hdr_greedy* hdr = HDR_GREEDY(p);
        switch (hdr->type_) {
            default:
                printf("Error with Greedy packet type. \n");
                exit(1);
        }
    } else {
        //ih->ttl_--;
        if (ih->ttl_ == 0) {
            drop(p, DROP_RTR_TTL);
            return;
        }
        forwardData(p);
    }
}
```

Fifth, we define forwardData(Packet* p) to handle packet forwarding. It checks if the destination is this node. If so, it will be forwarded to the upper layer by "port_dumux->recv(p, 0)". Otherwise, it will be forwarded to the lower layer by "send(p, 0)". Note that we define "getNextNode(Packet* p)" function that returns the closer neighbor to the destination. If the packet reaches the local minimum, the function returns -1 and then the packet is dropped.

```
void GreedyAgent::forwardData(Packet *p) {
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);

    if (ch->direction() == hdr_cmn::UP &&
        ((u_int32_t)ih->daddr() == IP_BROADCAST || ih->daddr() == addr_)) {
        port_dmux_->recv(p, 0);
        return;
    } else {
        /*
        / geedy fowarding
        */
        nsaddr_t target = getNextNode(p);
        if (target == -1) {
            Packet::free(p);
            return;
        }

        // handling forward the packet
        ch->direction() = hdr_cmn::DOWN;
        ch->addr_type() = NS_AF_INET;
        ch->next_hop_ = target;
        ch->last_hop_ = addr_;
        ch->num_forwards_++;

        send(p, 0);
    }
}
```

Although there are several local functions we need to write in the routing module, we already talk about the important parts in the routing module.

Modifying Existing C++ codes

To combine your original protocol with ns2, we have to modify packet/packet.cc as follow.

```
        // Bell Labs Traffic Trace Type (PackMime OL)
static const packet_t PT_BLTRACE = 60;

        // AOMDV packet
static const packet_t PT_AOMDV = 61;

        // Greedy forwarding packet
static const packet_t PT_GREEDY = 62;

        // insert new packet types here
static packet_t       PT_NTYPE = 63 // This MUST be the LAST one


         // XCP
         name_[PT_XCP]="xcp";

         // Bell Labs (PackMime OL)
         name_[PT_BLTRACE]="BellLabsTrace";

         // AOMDV patch
         name_[PT_AOMDV]= "AOMDV";

         name_[PT_GREEDY] = "GREEDY";

         name_[PT_NTYPE]= "undefined";
 }
```

In addition, we need to modify queue/priqueue.cc as follow.

```
void
PriQueue::recv(Packet *p, Handler *h)
{
        struct hdr_cmn *ch = HDR_CMN(p);

        if(Prefer_Routing_Protocols) {

                switch(ch->ptype()) {
                case PT_DSR:
                case PT_MESSAGE:
                case PT_TORA:
                case PT_AODV:
                case PT_GREEDY:
                // AOMDV patch
                case PT_AOMDV:
                        recvHighPriority(p, h);
                        break;

                default:
                        Queue::recv(p, h);
                }
        }
        else {
                Queue::recv(p, h);
        }
}
```

Modifying Tcl/Tk code

To bind C++ and Tcl/TK, we need to modify some existing tcl codes. Since the new routing protocol defines its original packet type, we need to modify tcl/lib /ns-packet.tcl as follow.

```
# Mobility, Ad-Hoc Networks, Sensor Nets:
        AODV    # routing protocol for ad-hoc networks
        Diffusion       # diffusion/diffusion.cc
        IMEP    # Internet MANET Encapsulation Protocol, for ad-hoc networks
        MIP     # Mobile IP, mobile/mip-reg.cc
        Smac    # Sensor-MAC
        TORA    # routing protocol for ad-hoc networks
        # AOMDV patch
        AOMDV
        GREEDY
# Other:
        Encap   # common/encap.cc
        IPinIP  # IP encapsulation
        HDLC    # High Level Data Link Control
```

We need to modify tcl/lib/ns-lib.tcl so that we can create routing module instance from TCL code as follow. wwe first add a few line to call a function "create-greedy-agent $node", and then we define the function to create routing module instance.

```
# basestation address setting
if { [info exist wiredRouting_] && $wiredRouting_ == "ON" } {
        $node base-station [AddrParams addr2id [$node node-addr]]
}
if {$rtAgentFunction_ != ""} {
        set ragent [$self $rtAgentFunction_ $node]
} else {
        switch -exact $routingAgent_ {
            GREEDY {
                    set ragent [$self create-greedy-agent $node]
            }
            DSDV {
                    set ragent [$self create-dsdv-agent $node]
            }
            DSR {
                    $self at 0.0 "$node start-dsr"
            }
            AODV {
                    set ragent [$self create-aodv-agent $node]
            }
```

```
# for Greedy
Simulator instproc create-greedy-agent { node } {
        set ragent [new Agent/Greedy [$node node-addr]]

        set addr [$node node-addr]

        $ragent addr $addr
        $ragent node $node
        if [Simulator set mobile_ip_] {
                $ragent port-dmux [$node set dmux_]
        }
        $node addr $addr
        $node set ragent_ $ragent

        $self at 0.0 "$ragent set-location"
        $self at 0.0 "$ragent start"
        return $ragent
}
```

When we want to access to C++ code from TCL, we need to add some function to tcl/lib/ns-mobilenode.tcl as follow. To be specific, we will add functions to access routing agent from your scenario file, to set up neighbors for a node, and to set up CBR flow information (this is for a source node to get a destination location).

```
Node/MobileNode instproc get-ragent {} {
        $self instvar ragent_
        return $ragent_
}

Node/MobileNode instproc set-flow-data {dest x y} {
        $self instvar ragent_
        $ragent_ set-flow-data $dest $x $y
}

Node/MobileNode instproc set-nbr {n x y} {
        $self instvar ragent_
        $ragent_ set-nbr $n $x $y
}
```

Compile

Before we compile, we have to modify Makefile.in. Otherwise, the source codes we have just written will not be compiled. We need to add .cc file to "OBJ_CC =" in Makefile.in as follow.

```
link/dynalink.o routing/rtProtoDV.o common/net-interface.o \
mcast/ctrMcast.o mcast/mcast_ctrl.o mcast/srm.o \
common/sessionhelper.o queue/delaymodel.o \
mcast/srm-ssm.o mcast/srm-topo.o \
routing/alloc-address.o routing/address.o \
$(LIB_DIR)int.Vec.o $(LIB_DIR)int.RVec.o \
$(LIB_DIR)dmalloc_support.o \
greedy/greedy.o greedy/greedy_nbr.o greedy/greedy_flow.o \
webcache/http.o webcache/tcp-simple.o webcache/pagepool.o \
webcache/inval-agent.o webcache/tcpapp.o webcache/http-aux.o \
webcache/mcache.o webcache/webtraf.o \
webcache/webserver.o \
webcache/logweb.o \
```