Implementation of Feedback Mechanism into AODV based on NS2

Sebastian Roschke
[sebastian.roschke@hpi.uni-potsdam.de]

2007-05-16

Abstract

This paper gives an overview on the implementation of a feedback mechanism into the AODV routing protocol based on NS2. This implementation work was done as part of seminar called *Attacks And Security Strategies in Mobile And Ad-hoc Networks*. The requirements of that project will be determined and the specific implementation details will be discussed afterwards. Some detailed information on NS2(e.g. structure, AODV and Scheduler) will be described as well, as it is necessary for understanding.

1 Introduction

The main goal of this project was the implementation of a feedback mechanism described in the paper *Feedback-based Solution for Avoiding Attacks on Mobile Ad-Hoc Networks*[1]. This implementation should be used for simulation of this mechanism in virtual ad-hoc networks via the well known simulation software *NS2*. The simulation results should be used for evaluation of that mechanism, e.g. in time overhead or data overhead issues.

2 Requirements

This chapter describes the required steps to achieve the described goal. The main subtasks to achieve this goal are:

- Implementation of feedback mechanism
- Implementation of malicious node behavior
- Implementation of simulation evaluation
- Development and Implementation of specific network simulations

The implementation of the feedback mechanism itself should be done on the routing layer. The mechanism itself works directly on the routing layer. It discovers malicious behavior on the routing layer. It sends additional messages that directly affect the behavior on the routing layer, e.g. no routing over untrusted or malicious nodes. The implementation can be done by modification of existing routing implementation; e.g. AODV. This will be described in section 4 of this paper.

The implementation of the malicious node behavior can be done on different layers, because of the different attack scenarios and vulnerabilities. In this specific example only one attack on routing layer is discussed. An malicious node performing an attack is expected here as a network node with selfish behavior. This node for example does not route packets from other nodes, but drops them. Based on this understanding of an attack, the implementation of the malicious node behavior should be done on the routing layer. The implementation can also be done by modification of existing routing implementation; e.g. AODV.

The implementation of the simulation evaluation can be done in a separate application apart from NS2. This application needs to read NS2 simulation output files, to analyze them and to visualize the results. This will not be described within this paper.

All the implementation work is tested with specific NS2 simulation scripts. These scripts are based on TCL and describes the simulation itself; e.g. the size of the simulation area, the number of the nodes within the area and the number and type of the packets send during this simulation. A useful documentation of NS2 simulation scripts is NS-By-Example[2].

3 Technical Background on NS2

This section describes the technical background of the implementation work. It covers specific parts of the NS2 implementation. More precisely an overview over NS2 is provided, the functionality of the scheduler and the implementation of the AODV implementation will be described. The basis for this implementation is NS2 version 2.30. The implementation is done with the NS2 all-in-one package[3, 4]. The information provided in this section can be found within documentation and the source code. The investigations of AODV protocol functionality are made by NS2 debugging[5].

3.1 Structure

3.1.1 Class Hierarchies

NS2 is a network simulator written in C++ and TCL. The NS2 front end is an OTCL interpreter. The simulation descriptions are scripts written in OTCL. The front end interprets this files and the back end simulates the described scenario. The NS2 simulation output files are generated within this process. Within NS2 there are two separate class hierarchies; the *Compiled Hierarchy* implemented in C++ and the *Interpreted Hierarchy* implemented in TCL. Furthermore there are classes that realize the processing of the simulation and there are classes that belong to the simulation itself. Within this document objects based on classes that realize the processing of the simulation are called *Processing Objects*. The Objects based on the classes that belong to the simulation itself will be called *Simulation Objects*. Most of the Processing Objects are implemented in the Compiled Hierarchy. For most of the Simulation Objects, there is 1:1 relationship between a base class in the Compiled Hierarchy and a base class in the Interpreted Hierarchy. The implementation of this classes is often separated in two parts, one located in the Compiled Hierarchy and one located in the Interpreted Hierarchy. This makes modification of Simulation Objects very complicated.

The separation in two class hierarchies is reasonable for dealing with the trade-off between runtime performance and iteration time; which means the required time to change the simulation descriptions and execute another simulation. The iteration time can be minimized by utilization of simple scripting language with simple syntax. The runtime performance can be optimized by using a compiled language. Therefore the two approaches are combined within NS2 by using OTCL as front end and C++ as back end.

3.1.2 TCL Linkage

The *TCL Linkage* is the communication interface between the implementation of the Simulation Objects on the two class hierarchies; the Compiled Hierarchy and the Interpreted Hierarchy. It realizes the communication between the implementation parts; e.g. the transmission of messages from Compiled Hierarchy into Interpreted Hierarchy and vice versa. The TCL Linkage consists of six important classes:

- Tcl
- TclObject
- TclClass
- TclCommand
- EmbeddedTcl
- InstVar

The functionality of the TCL Linkage is described in chapter 3 of the NS2 documentation[4].

3.1.3 File System Structure

Figure 1 shows the structure of the file system within NS2. Only the directories are shown, that have a meaning for the described implementation.

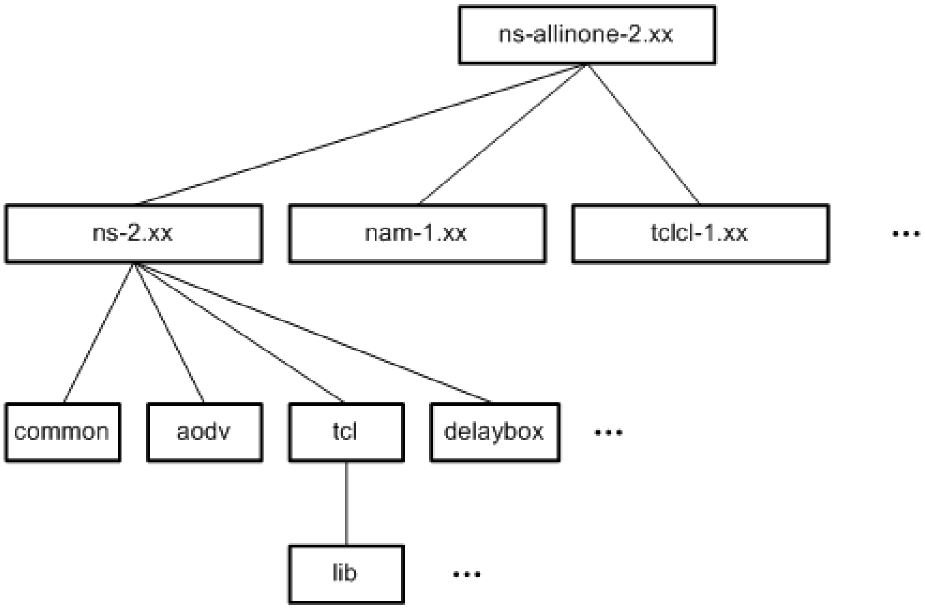

Figure 1: File System Structure

- The root directory *˜/ns-allinone-2.30/* includes the complete NS2 system.
- The TCL interpreter directory ˜/ns-allinone-2.30/tclcl/ includes all files related to the TCL interpreter.
- The NS2 simulator directory *˜/ns-allinone-2.30/ns-2.30/* includes all files related to the simulator. This includes the class implementations on Compiled Hierarchy and on Interpreted Hierarchy.
- The NS2 Processing Objects directory *˜/ns-allinone-2.30/ns-2.30/common/* includes all class implementations on Compiled Hierarchy.
- The NS Processing Objects directory *˜/ns-allinone-2.30/ns-2.30/tcl/* includes the class implementations on Interpreted Hierarchy.
- The NS Simulation Objects directory *˜/ns-allinone-2.30/ns-2.30/tcl/lib/* includes the class implementations on Interpreted Hierarchy.
- The AODV directory *˜/ns-allinone-2.30/ns-2.30/aodv* includes all classes used by AODV implementation.

Figure 2 shows the most important classes in NS2 and can be used as overview.

3.2 Scheduler

The Scheduler is an entity within NS2 responsible for packet transmission. The scheduler is a single instance within the NS2 runtime that collects and transmits

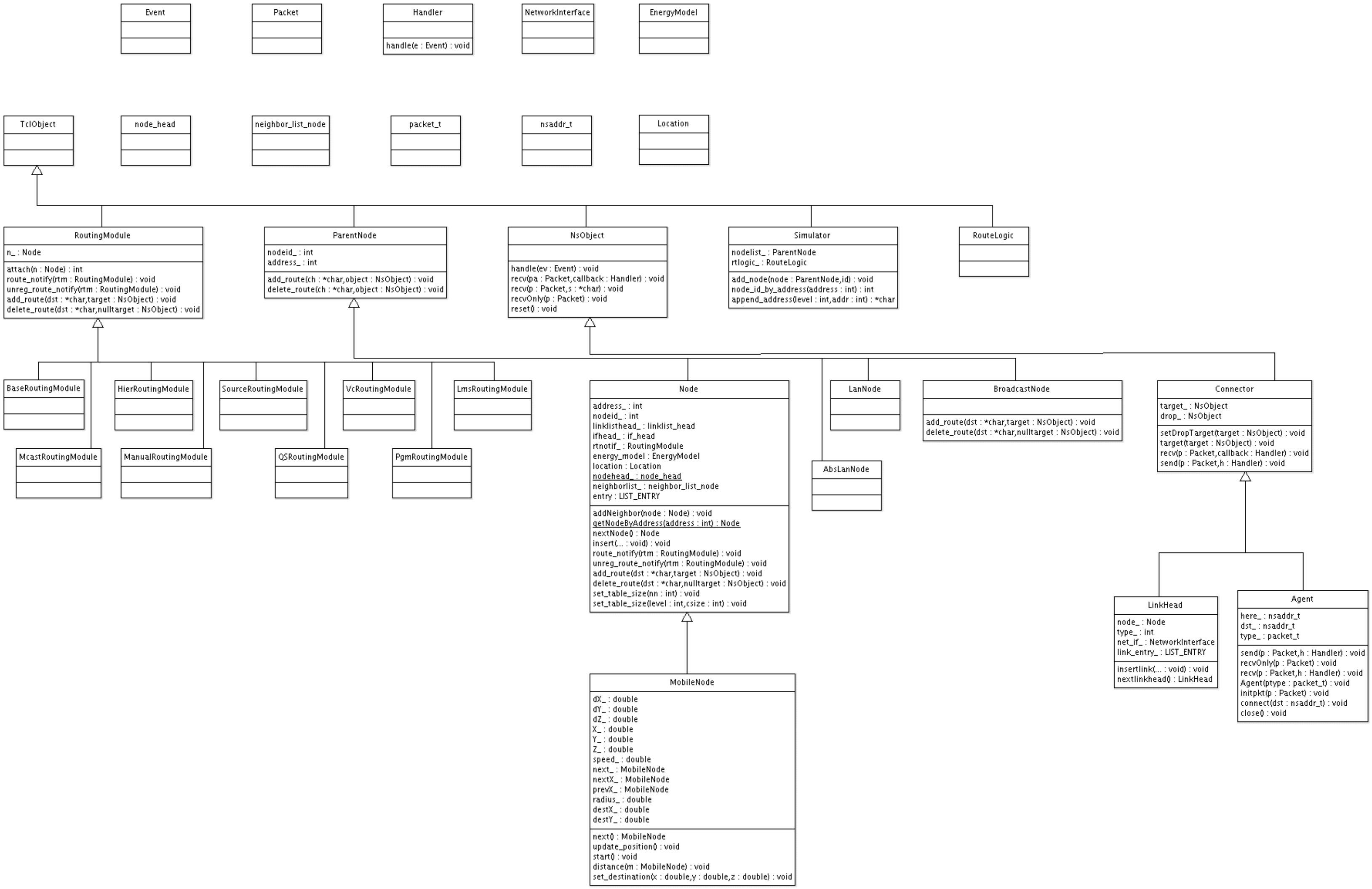
Event
Packet
Handler
handle(e : Event) : void
NetworkInterface
EnergyModel
TclObject
node_head
neighbor_list_node
packet_t
nsaddr_t
Location
RoutingModule
n_ : Node
attach(n : Node) : int
route_notify(rtm : RoutingModule) : void
unreg_route_notify(rtm : RoutingModule) : void
add_route(dst : *char,target : NsObject) : void
delete_route(dst : *char,nulltarget : NsObject) : void
ParentNode
nodeid_ : int
address_ : int
add_route(ch : *char,object : NsObject) : void
delete_route(ch : *char,object : NsObject) : void
NsObject
handle(ev : Event) : void
recv(pa : Packet,callback : Handler) : void
recv(p : Packet,s : *char) : void
recvOnly(p : Packet) : void
reset() : void
Simulator
nodelist_ : ParentNode
rtlogic_ : RouteLogic
add_node(node : ParentNode,id) : void
node_id_by_address(address : int) : int
append_address(level : int,addr : int) : *char
RouteLogic
BaseRoutingModule
HierRoutingModule
SourceRoutingModule
VcRoutingModule
LmsRoutingModule
McastRoutingModule
ManualRoutingModule
QSRoutingModule
PgmRoutingModule
Node
address_ : int
nodeid_ : int
linklisthead_ : linklist_head
ifhead_ : if_head
rtnotif_ : RoutingModule
energy_model : EnergyModel
location : Location
nodehead_ : node_head
neighborlist_ : neighbor_list_node
entry : LIST_ENTRY
addNeighbor(node : Node) : void
getNodeByAddress(address : int) : Node
nextNode() : Node
insert(... : void) : void
route_notify(rtm : RoutingModule) : void
unreg_route_notify(rtm : RoutingModule) : void
add_route(dst : *char,target : NsObject) : void
delete_route(dst : *char,nulltarget : NsObject) : void
set_table_size(nn : int) : void
set_table_size(level : int,csize : int) : void
LanNode
AbsLanNode
BroadcastNode
add_route(dst : *char,target : NsObject) : void
delete_route(dst : *char,nulltarget : NsObject) : void
Connector
target_ : NsObject
drop_ : NsObject
setDropTarget(target : NsObject) : void
target(target : NsObject) : void
recv(p : Packet,callback : Handler) : void
send(p : Packet,h : Handler) : void
MobileNode
dX_ : double
dY_ : double
dZ_ : double
X_ : double
Y_ : double
Z_ : double
speed_ : double
next_ : MobileNode
nextX_ : MobileNode
prevX_ : MobileNode
radius_ : double
destX_ : double
destY_ : double
next() : MobileNode
update_position() : void
start() : void
distance(m : MobileNode) : void
set_destination(x : double,y : double,z : double) : void
LinkHead
node_ : Node
type_ : int
net_if_ : NetworkInterface
link_entry_ : LIST_ENTRY
insertlink(... : void) : void
nextlinkhead() : LinkHead
Agent
here_ : nsaddr_t
dst_ : nsaddr_t
type_ : packet_t
send(p : Packet,h : Handler) : void
recvOnly(p : Packet) : void
recv(p : Packet,h : Handler) : void
Agent(ptype : packet_t) : void
initpkt(p : Packet) : void
connect(dst : nsaddr_t) : void
close() : void

Figure 2: NS2 Class Tree

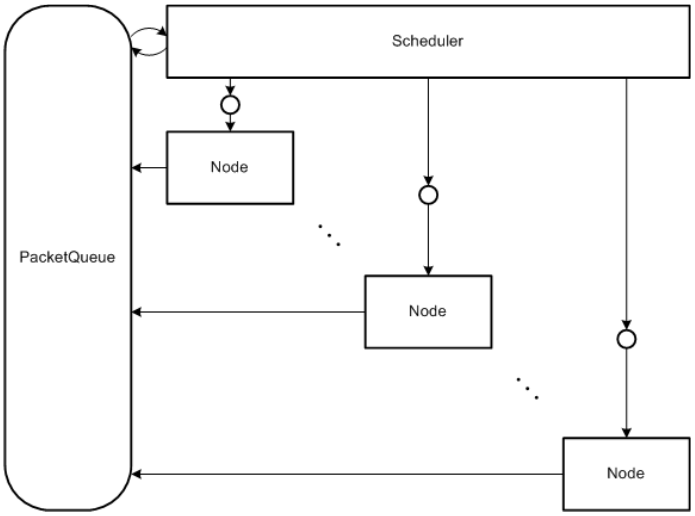

Figure 3: Scheduler

messages. Therefore the Scheduler class is realized as static class with static methods.

Figure 3 shows the logical structure of the scheduler within NS2. The Scheduler consists of control unit and a so called packet queue. This packet queue is accessible through the nodes within the simulation. Each node can put packets into the packet queue. The Scheduler takes each packet from the queue and dispatches the packet to its destination. The process is shown in figure 4. The knows the destination of a packet during runtime. The destination of the packet is saved within a special pointer in the packet structure. This pointer is a reference to a *NSObject* instance where the method *recv()* is called within *Scheduler::dispatch()*.

3.3 AODV Implementation

In mobile and ad-hoc networks there are several routing protocols available; e.g. DSR, DSDV or AODV. Within NS2 there are implementations for some of the available routing protocols. The AODV implementation in NS2 is the basis for the implementation work and is therefore explained in this section. This AODV implementation is not RFC compliant; e.g. there is no implementation for the blacklist defined in AODV[6]. The AODV implementation mainly consists of two functional units.

- Mechanism to detect the route through a network
- Mechanism to forward packets within this route

Figure 5 a simple packet flow within a single node in the simulation. An agent consists to specific node. This agent allocates the packet and initializes the

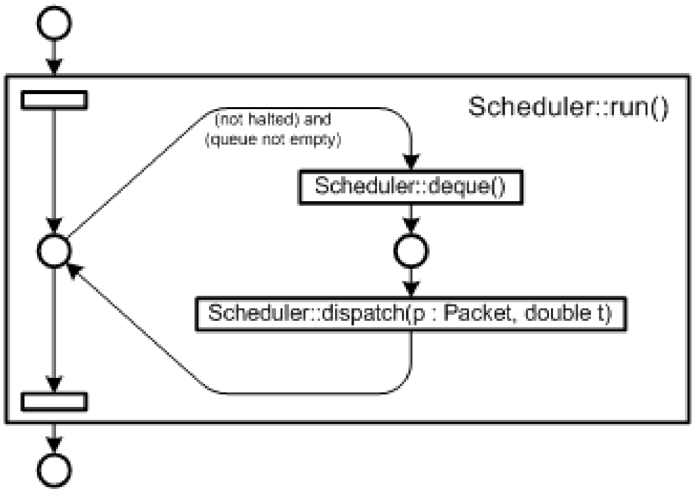

Figure 4: Scheduler Process

content of the packet. Afterwards the packet is handled by the AODV routing mechanism. First the routing handler receives this packet. This is done locally. Afterwards the route for the packet is detected. When a suitable route is found, the packet is forwarded to the next hop within this route. The forward is done by the Scheduler described in section 3.

This figure shows the sequence of calls within the software objects. The call hierarchy in AODV is as follows. The recv() method is the first method called. This method calls first the resolve() and the forward() method. The forward() method calls the Scheduler::scheduler() method.

Figure 6 shows the initialization process of the AODV implementation in a specific simulation. This simulation consists of three hosts that are located in a line, so that *Host 0* and *Host 2* are not able to communicate directly. Within this simulation one TCP packet is send from Host 0 to Host 2 and the reply is send from Host 2 to Host 0. AODV resolves the route to Host 2 by sending a special *request packet*. This packet is broadcasted as long as the destination of the packet replies. The reply is also broadcasted through the network. Each forwarded request packet also triggers the intermediate nodes to send additional request packets.

Therefore in the beginning of the simulation there is a high amount of AODV related network traffic. After that phase, there is only a few additional AODV network traffic required. Detected routes are cached within this implementation, so that not every send packets requires a related AODV route request. After the detection of the route, the packet is send.

The the packet flow during the transmission is relatively simple. Figure 7 shows the process with the related calls. The packet is forwarded by each node on the route. In the forward method the Scheduler is called, to transmit the packets. The resolve() method returns immediately, due to a cached response

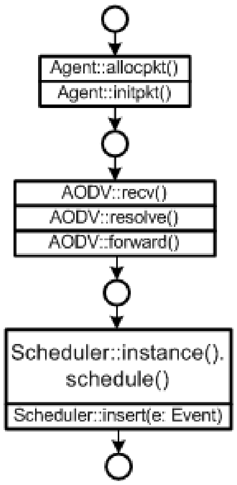

Figure 5: Packet Flow

for the request.

The figure 8 shows the class structure of AODV related classes.

4 Implementation

This sections covers the main ideas of the implementation work. Based on the technical background the implementation is done by modification of the existing mechanisms.

4.1 Feedback Mechanism

This implementation is done by modification of the existing AODV implementation. This implementation is described in section 3.3. To implement the feedback mechanism, the following issues are realized.

The header structure of the common packet is changed. The header hdr_cmn is the main header structure utilized for each packet in NS2. The common header is extended with the following information:

```
bool recv_fb_req;
bool one_hop_fb_req;
bool two_hop_fb_req;

int fb_type;
int fb_dst;
int fb_path[PATH_LEN];
```

The first three members are boolean values utilized to encode the requested feedback into the packet. Each packet needs to carry information for it's required

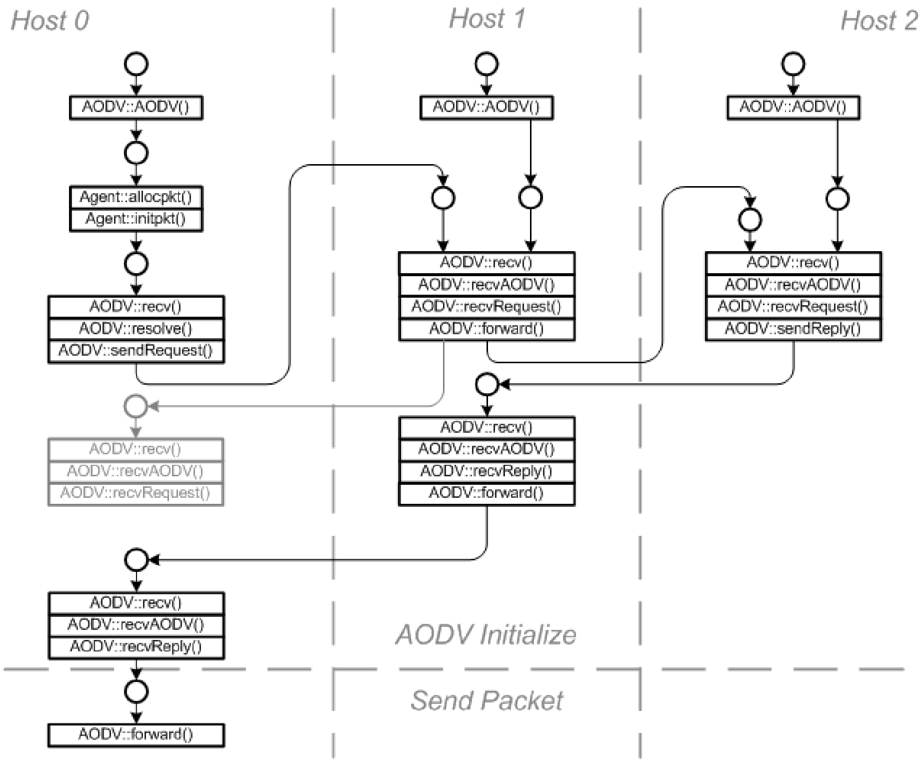

Figure 6: AODV Initialization

feedback. The integer members fb_type and fb_dst are used for feedback packets to encode the type of the feedback and it's destination. The integer fb_path array fb_path is used to save the feedback route.

Additional packet types are implemented, utilized by nodes to transmit information according to the feedback mechanism. The following packet types are added:

- FB
- FB_REQ
- FB_ACK
- FB_DEV

The FB packet is a general feedback packet. This packet will be send when a packet is received and a feedback is required. The FB_REQ packet is used to request a feedback of a host that was not send in time. The FB_ACK packet is used to acknowledge a received feedback. The FB_DEV packet is used to inform neighbor nodes when the reputation of a node is devalued.

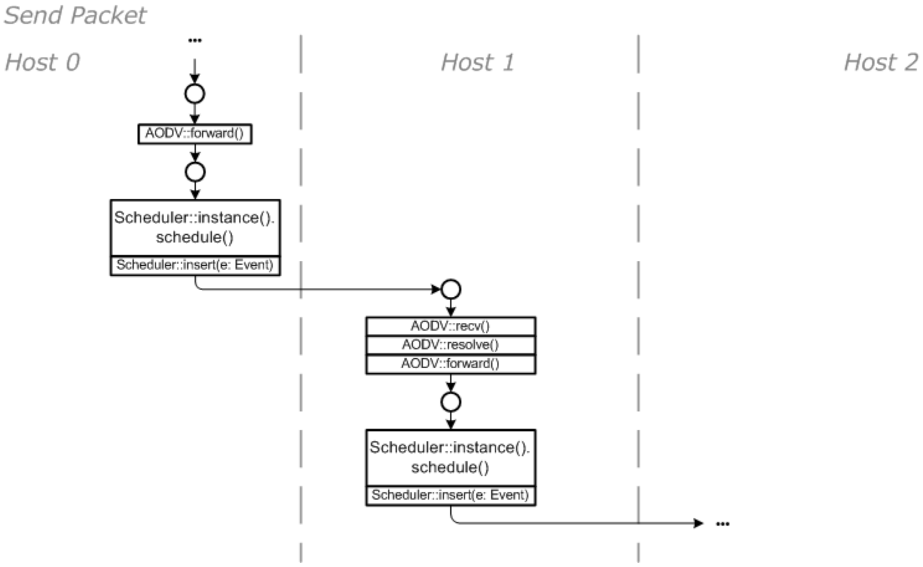

Figure 7: Transmission Packet Flow

Several new data storages are implemented within the class AODV. The following data storages related to the feedback mechanism are implemented:

```
list<ReputationListElement>     repList;
list<ReputationListElement>     sendFeedbackList;
list<ForwardedListElement>      forwardedPacketList;
```

The repList is used to save all reputation values of the neighbor nodes. The sendFeddbackList saves all the feedback packets that the node sends. The forwardedPacketList saves all forwarded packets, processed by this host.

To send new types of packets there is a new timeout handler required. This timeout handler get a callback, when he timeout is reached. When there is no feedback acknowledge received yet, the feedback is requested. If this fails, the reputation of the neighbor host, responsible for the feedback, is decreased.

The AODV class is extended with methods to send and receive feedback packets. The following methods are added:

```
void sendFB(Packet *pfb,nsaddr_t dst, int uid,int type);
void sendFBRequest(nsaddr_t dst);
void sendFBAck(nsaddr_t dst);
void sendFBDevalue(nsaddr_t dst);
void recvFB(Packet *p);
void recvFBRequest(Packet *p);
void recvFBAck(Packet *p);
void recvFBDevalue(Packet *p);
```

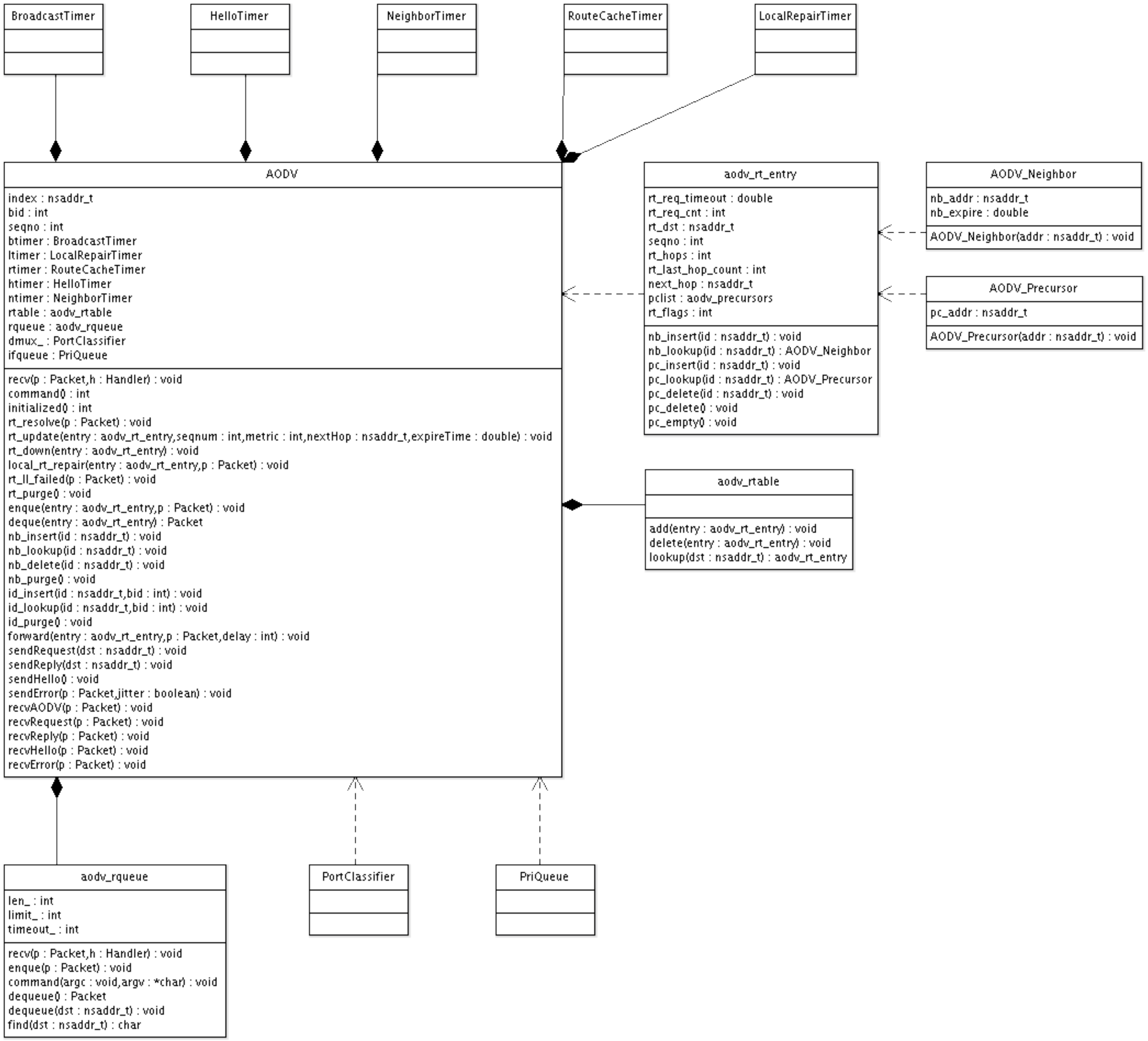
BroadcastTimer
HelloTimer
NeighborTimer
RouteCacheTimer
LocalRepairTimer
AODV
index : nsaddr_t
bid : int
seqno : int
btimer : BroadcastTimer
ltimer : LocalRepairTimer
rtimer : RouteCacheTimer
htimer : HelloTimer
ntimer : NeighborTimer
rtable : aodv_rtable
rqueue : aodv_rqueue
dmux_ : PortClassifier
ifqueue : PriQueue
recv(p : Packet,h : Handler) : void
command() : int
initialized() : int
rt_resolve(p : Packet) : void
rt_update(entry : aodv_rt_entry,seqnum : int,metric : int,nextHop : nsaddr_t,expireTime : double) : void
rt_down(entry : aodv_rt_entry) : void
local_rt_repair(entry : aodv_rt_entry,p : Packet) : void
rt_ll_failed(p : Packet) : void
rt_purge() : void
enque(entry : aodv_rt_entry,p : Packet) : void
deque(entry : aodv_rt_entry) : Packet
nb_insert(id : nsaddr_t) : void
nb_lookup(id : nsaddr_t) : void
nb_delete(id : nsaddr_t) : void
nb_purge() : void
id_insert(id : nsaddr_t,bid : int) : void
id_lookup(id : nsaddr_t,bid : int) : void
id_purge() : void
forward(entry : aodv_rt_entry,p : Packet,delay : int) : void
sendRequest(dst : nsaddr_t) : void
sendReply(dst : nsaddr_t) : void
sendHello() : void
sendError(p : Packet,jitter : boolean) : void
recvAODV(p : Packet) : void
recvRequest(p : Packet) : void
recvReply(p : Packet) : void
recvHello(p : Packet) : void
recvError(p : Packet) : void
aodv_rt_entry
rt_req_timeout : double
rt_req_cnt : int
rt_dst : nsaddr_t
seqno : int
rt_hops : int
rt_last_hop_count : int
next_hop : nsaddr_t
pclist : aodv_precursors
rt_flags : int
nb_insert(id : nsaddr_t) : void
nb_lookup(id : nsaddr_t) : AODV_Neighbor
pc_insert(id : nsaddr_t) : void
pc_lookup(id : nsaddr_t) : AODV_Precursor
pc_delete(id : nsaddr_t) : void
pc_delete() : void
pc_empty() : void
AODV_Neighbor
nb_addr : nsaddr_t
nb_expire : double
AODV_Neighbor(addr : nsaddr_t) : void
AODV_Precursor
pc_addr : nsaddr_t
AODV_Precursor(addr : nsaddr_t) : void
aodv_rtable
add(entry : aodv_rt_entry) : void
delete(entry : aodv_rt_entry) : void
lookup(dst : nsaddr_t) : aodv_rt_entry
aodv_rqueue
len_ : int
limit_ : int
timeout_ : int
recv(p : Packet,h : Handler) : void
enque(p : Packet) : void
command(argc : void,argv : *char) : void
dequeue() : Packet
dequeue(dst : nsaddr_t) : void
find(dst : nsaddr_t) : char
PortClassifier
PriQueue

Figure 8: AODV Class Structure

The trace subsystem of NS2 is modified, due to the additional packet types. The new packet types are processed and logged through this logging system.

4.2 Malicious Node Behavior

The malicious node behavior here only covers the packet dropping within the route. It is a selfish behavior of a node to increase it's network performance. To implement the attack on the routing layer the AODV forwarding mechanism is modified in the following way.

A node is configured to show malicious behavior through a configuration file. When the AODV object is instantiated the first time, a file is read that includes all malicious node configurations. The AODV class holds a static list to save the node configurations.

```
static list<MaliciousNodeListElement>   maliciousNodeList;
```

Each node is configured through the members badNode and dropping_rate. The boolean member bad node holds information whether the node is malicious. The dropping_rate member holds the information of the dropping rate of this malicious node.

```
bool badNode;
int dropping_rate;
```

The nodes configured as malicious nodes are dropping forwardable packets with a specified rate. This is done by modification of the forward method within AODV. The dropping of a packet is logged through trace subsystem.

4.3 Blacklist

As described in section 3.3 the AODV implementation is not RFC compliant. The blacklist is needed by the feedback mechanism to avoid detected malicious nodes. The current AODV implementation does not provide a working blacklist. Therefore the blacklist is newly implemented into AODV by modification of the AODV class.

```
list<nsaddr_t> blacklist;
```

The blacklist member is added into the AODV class. Each host listed on the blacklist is excluded from the AODV routing. Request and replies from those hosts are not accepted and the forwarding of packets from this hosts is disabled.

References

[1] *Feedback-based Solution for Avoiding Attacks on Mobile Ad-Hoc Networks*, 2007.

[2] Ns2 by example. [Online]. Available: http://nile.wpi.edu/NS/

[3] Ns2 source. [Online]. Available: http://www.isi.edu/nsnam/dist/ns-allinone-2.30.tar.gz

[4] Ns2 documentation. [Online]. Available: http://www.isi.edu/nsnam/ns/doc/index.html

[5] Ns2 debugging. [Online]. Available: http://www.cs.ust.hk/ cszyz/ns2-debug.html

[6] Ad hoc on-demand distance vector (aodv) routing. [Online]. Available: http://www.ietf.org/rfc/rfc3561.txt