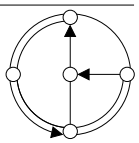
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



WS 2005-2006

Roger Wattenhofer, Nicolas Burri, Pascal von Rickenbach,
Yves Weber, and Andreas Wetzel

Mobile Computing
Exercise 3

Assigned: November 21, 2005
Due: December 05, 2005

Dynamic Source Routing

One of the most interesting issues in mobile ad hoc networks is multihop routing: If two devices wishing to communicate cannot hear each other directly, intermediate nodes will relay their messages. In this exercise we will implement such a routing algorithm, in particular a variant of Dynamic Source Routing (DSR).

The basic idea of DSR consists in the use of source routes: The communication source attaches a predefined route towards the destination to each packet it sends; the packet is afterwards sent along this route, intermediate nodes simply bouncing the packet to the next terminal found in the route. This part of the algorithm is sometimes referred to as the **forwarding phase** of DSR.

But how does the source know about a valid route to the destination? This question is answered in what is usually called **route discovery**. The route discovery phase consists of a **flooding** and a **reply** stage. The source initiates route discovery by broadcasting a *route request* message. All nodes (other than the destination) receiving such a message for the first time attach their address to the route in the packet and rebroadcast this message as long as the TTL value allows so. The destination receiving a *route request* answers by sending a *route reply* message to the original node using the reverted route from the received *route request* message.[1] If the original source receives a *route reply* message, it can store the discovered route in a cache and use it to send subsequent (user) messages to the according destination.[2]

This is a completely demand-driven routing algorithm. Messages are only transmitted as necessary: There is no periodic message exchange. A source wishing to send a (user) message, first checks whether a route to the destination is already stored in its route cache. If so, the packet is equipped with the route and transmitted. The destination receiving the message will reply with an acknowledge message, reverting the route from the received message. If this message arrives back at the original sender, the latter can report successful transmission of the initial message. If however there is no cached route or if the message is not acknowledged within a certain timeout (for instance due to moving network nodes), the source will initiate route discovery.

As a first step we are going to implement the flooding facility in this exercise. We therefore define a message type: *route request* (RREQ). The following table defines the packet format for this message (of course you still need an additional packet header for the "single hop" layer, consisting of a sender and a receiver field).

[1] Note that this mechanism works only if links are symmetric. For the sake of simplicity we assume so for this exercise.

[2] It is possible to implement several optimizations for this algorithm. For instance, already an intermediate node other than the destination could reply with a *route reply* message provided that it knows of a route to the destination. Furthermore it would also be possible to cache more than one route for a destination in order to postpone route discovery in case of route failure. Also the use of implicit acknowledgements (Lecture 5) would be possible. We suggest to first implement the basic algorithm; optimizations can later be added.

Message Type	Message Format (field size in bytes)
RREQ	type (1) | flood ID (2) | sender (2) | receiver (2) | TTL (1) | route length (1) | route (variable)

The type field contains the message type value (see below). The flood IDs are necessary to match corresponding messages and to control flooding: Only a *route request* containing a (sender, flood ID) pair seen for the first time should be rebroadcasted. The IDs are unsigned 16 bit numbers which should be generated increasingly and wrap around (start again from 0) when reaching `0xffff`.[3] The sender and receiver fields contain the addresses of the source and the destination of the complete route. The end of each packet is formed by the complete route (including source and destination), whose format consists of a sequence of 2 byte-addresses, starting with the source. Correspondingly the message contains a route length field describing the length of the complete route and a TTL field as discussed above.

The following table defines the message type values and summarizes the way a routing node *must* react upon receipt of a RREQ message.

Message Type	Type Value	Reaction upon Receipt
RREQ	0x32	If TTL > 1, decrement TTL, append my address to the route, and rebroadcast; if TTL $= 1$, do not rebroadcast; if TTL $= 0$, rebroadcast leaving TTL unchanged (flood complete network)

The "multi hop" functionality should be implemented within a separate communication layer. It will be placed on top of the "single hop" layer from Exercise 2 and should provide the usual functionality to possible higher layers.

In order to test your implementation, you should print the route from source to destination once the packet reaches its target client.

[3] In order to cope with this wrap-around (and also relaunched applications—which therefore restart to increment their IDs at 0), a perfect implementation should contain some kind of "ID ageing", having "old" IDs lose their validity. In a preliminary version this problem can however be neglected.