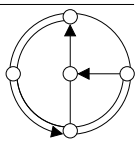
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



SS 2003 Roger Wattenhofer, Keno Albrecht, Aaron Zollinger

Mobile Computing
Exercise 5

Assigned: June 5, 2003
Due: June 26, 2003

1 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**. For this exercise we will use controlled flooding to find routes. According to trick 1 of Lecture 5, the flooding messages will contain a TTL (time to live) value decremented at each intermediate node, which prevents the whole network from unnecesseraly being flooded in cases where the destination is relatively close to the source.

The route discovery phase itself 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. We suggest to first broadcast a *route request* message with TTL 1 in order to detect a neighboring destination without flooding the network. In case of failure, again after a certain timeout, retry by sending *route request* messages with TTL values 2, 4, and 8, as long as necessary. If still no route has been discovered, finally set TTL to 0, which will lead to flooding of the whole network. If again there is no positive answer, we give up and report an error message.

As in the last exercise we define different message types: *source route message* (SRMSG), *source route acknowledgement* (SRACK), *route request* (RREQ), and *route reply* (RREP). The following table defines the packet formats for these messages.

[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)
SRMSG	type (1) | message ID (2) | sender (2) | receiver (2) | data length (2) | route length (1)| route index (1) | data (variable) | route (variable)
SRACK	type (1) | message ID (2) | sender (2) | receiver (2) | route length (1) | route index (1) | route (variable)
RREQ	type (1) | flood ID (2) | sender (2) | receiver (2) | TTL (1) | route length (1) | route (variable)
RREP	type (1) | flood ID (2) | sender (2) | receiver (2) | route length (1) | route index (1) | route (variable)

The type field contains the message type value (see below). The message and flood IDs are necessary to match corresponding messages and acknowledgements and to control flooding: Only a *route request* containing a (sender, flood ID) pair seen for the first time should be rebroadcasted or answered to with an RREQ message, respectively. 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. SRMSG contains the user data (of variable size) and accordingly a data length field. 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 all messages contain a route length field describing the length of the complete route. Except for RREQ, all messages contain a route index field pointing to the next node a message will be forwarded to. For completeness, RREQ contains a TTL field as discussed above.

The following table defines the message type values and summarizes the ways a routing node *must* react upon receipt of a message of according type. Again, in some cases additional local action will be useful, if not necessary.

Message Type	Type Value	Reaction upon Receipt
SRMSG	0x30	if the destination is reached, reply with an SRACK; otherwise forward the message to the next node in the route
SRACK	0x31	if the destination is not reached yet, forward
RREQ	0x32	if the destination is reached, reply with an RREP; otherwise: 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)
RREP	0x33	if the destination is not reached yet, forward

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

Finally some hints: Choose the timeouts according to the maximum number of hops in a roundtrip (message $\rightarrow$ acknowledgement). Find a reasonable timeout for the "final" route discovery try (TTL $=$ 0).

Up to now we have used a `receive()` method to receive packets. For this exercise, try to implement the same functionality using callbacks (a.k.a. listeners). This will simplify the implementation of the routing algorithm within a separate communication layer.

As in Exercise 4, there will be several cooperating threads. Again be sure to synchronize where necessary.

2 Multihop Instant Messenger

Adjust the instant messenger application implemented in Exercise 4 to use the above routing layer. Try to modify the messenger application in as few places as possible.

Ensure that the routing layer also passes received broadcast SRMSG packets to the application. Otherwise the messenger application will not be able to explore its neighborhood. In particular, (1) add the route [own address, broadcast address (= 0)] to your route cache and (2) neither acknowledge packets sent to the broadcast address nor expect an acknowledgement from such messages.

Add a "buddy list" to your instant messenger application. Here you can enter your friends' addresses and communicate with them even if they are more than one hop away!

[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.