Diploma Thesis

Implementing XTC on TinyOS

Yves Weber
webery@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer 2005

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Nicolas Burri & Pascal von Rickenbach

Abstract

The XTC algorithm is a powerful yet easy to understand topology control algorithm for wireless sensor networks. Since it does not require any knowledge about the nodes' positions but operates on a general notion of link qualities, it is very suitable for the use in small embedded sensor architectures.

This thesis provides an implementation of an extended version of the XTC algorithm for the mica2 sensor nodes running as an application on top of the TinyOS operating system. The algorithm has been extended to produce correct results in a dynamic environment as opposed to the original algorithm which is only suitable in a static scenario.

Besides the implementation of the XTC algorithm, this thesis also includes the realization of a dynamic source routing algorithm that can be run either directly on the network or on the topology created by XTC. By juxtaposing the results of the two options we illustrate the benefit of the XTC algorithm.

Contents

Chapter 1

Introduction

1.1 Topology Control in Sensor Networks

A sensor network consists of multiple small sensors which are connected by wireless radio. The sensors are typically rather small and feature — besides the radio module — a processor, some memory and a power source. If two sensors are in the transmission range of each other, these two sensors are called linked[1]. The sensors and their links form the network graph G.

Topology control algorithms create a topology graph G_{tc} which is a subgraph of G that meets several requirements. The goal is that the algorithm comes up with a subgraph that only contains good links. In general, good links are supposed to be short and energy efficient, such that all nodes are able to drop their long-range neighbors. As a result, the nodes can lower their transmission power and therefore save energy and reduce interference at the same time.

Such a resulting topology should provide the following properties:

- **Symmetry:** If a node u decides to maintain a connection to its neighbor v, then v keeps its link to u.

- **Connectivity:** If there is a connection from a node u to a node v in G (possibly using multiple nodes as relay stations), there is also a path from u to v in G_{tc}.

- **Sparseness:** G_{tc} is sparse, that is, the number of links in G_{tc} is in the order of the number of nodes.

Depending on the application, other requirements may be added and the properties above may be strengthened. For example, the sparseness property imposes a maximal node degree in G_{tc} instead of an asymptotic limit.

[1]Asymmetric links (i.e. one node is in the transmission range of another one, but not vice versa) are often ignored since even sending a simple acknowledgement message may become unacceptably complicated [6]

XTC Algorithm

```
1: Establish order $\prec_u$ over $u$'s neighbors in $G$
2: Broadcast $\prec_u$ to each neighbor in $G$; receive orders from all neighbors
3: Select topology control neighbors:
4:     $N_u := \{\}$; $\widetilde{N}_u := \{\}$
5:     while  ($\prec_u$ contains unprocessed neighbors) {
6:        $v :=$ least unprocessed neighbor in $\prec_u$
7:        if  $(\exists\, w \in N_u \cup \widetilde{N}_u : w \prec_v u)$
8:           $\widetilde{N}_u := \widetilde{N}_u \cup \{v\}$
9:        else
10:          $N_u := N_u \cup \{v\}$
11:    }
```

1.2 The XTC Algorithm

The XTC[2] algorithm [9] is a topology control algorithm which is very effective and easy to understand, but nevertheless exhibits impressive results. Depending on the model under consideration (e.g. the Unit Disk Graph [2]), different properties of the resulting topology graph G_{xtc} can be proven. For a theoretical treatment of the XTC algorithm, we refer to [9].

The XTC algorithm bases on an abstract link quality. The metrics used to determine the link quality can be chosen depending on the application — there are no restrictions from the algorithm. Possible metrics include but are not limited to signal strength, delay, error rate, or a combination of the former.

The box above shows the pseudocode of the XTC algorithm. It consists of three steps:

1. *Line 1*
   Each node u builds up a list of all its neighbors. This list is sorted according to the link quality in descending order. The list is called the order of u throughout the rest of this work.

2. *Line 2*
   Each node broadcasts its order to all its neighbors. It follows that all nodes receive the orders of all their neighbors.

3. *Lines 3-11*
   In the last step, a node decides which links it wants to keep active and which links are dropped. This decision is only based on the orders of the neighbors and its own order — there is no further communication required in this step.

   This step starts by initializing the two sets N_u and $\widetilde{N}_u$ to be empty. N_u will be filled with the neighbors of u in G_{xtc} (i.e. the links which are kept active by XTC) while $\widetilde{N}_u$ will contain the neighbors of u to which no

[2]The abbreviation XTC — while the pronounciation is unambiguous — has not a definitive meaning yet. TC stands for topology control. Candidates for the X include ”exotic”, ”extreme”, ”exceptional”, and others.

direct link will be maintained in G_{xtc}. After initializing these two sets, the algorithm traverses all neighbors of u in increasing order and decides according to the criteria in line 7 whether to add the neighbor to N_u or $\widetilde{N}_u$.

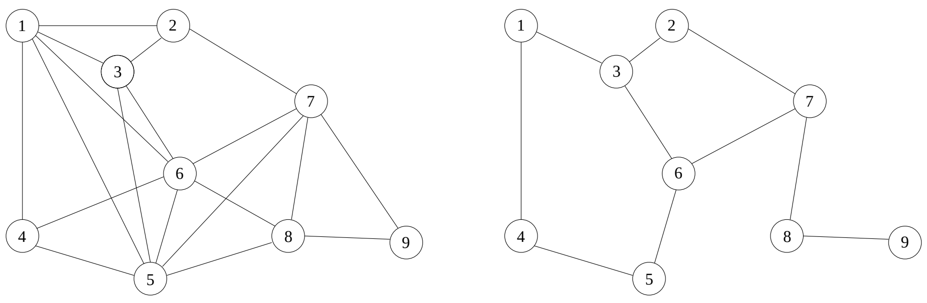


Figure 1.1: A sensor network graph G (left) and the result G_{xtc} of the XTC algorithm when we use the Euclidian distance as link quality metric (right).

Figure 1.1 shows a possible graph of a sensor network at the left. On the righthand side, the topology graph is displayed resulting from the application of the XTC algorithm when using the Euclidian distance as link quality metric to build up the order at each node.

1.3 mica2 Motes

1.3.1 Hardware

The sensor nodes used in this diploma thesis are the mica2 motes developed by Crossbow Technology Inc. [3]. These sensors are widely used for research projects and offer a broad collection of sensor boards. A mica2 mote with two accessories — a sensor board for data acquisition and an ethernet gateway — are shown in Figure 1.2.

The mica2 motes feature the following basic data:

Processor:	8 bit ATMega128L processor running at 7.37 MHz
Memory:	128 kB program memory, 4 kB SRAM, 512 kB EEPROM
Radio Interface:	ChipCon CC1000 chip sending at a frequency of 433 or 900 MHz respectively, allowing data rates of up to 38.4 kbps
Power Source:	2AA (1.2 V)
Size:	58 x 32 x 7 mm (without batteries)
Weight:	200 g (with batteries)
Misc.:	3 LEDs (green, yellow, and red), 51-pin expansion connector allowing to connect external peripherals

1.3.2 Software

The TinyOS [7] operating system running on the motes was originally developed at the University of Berkeley. It is an event based operating system designed

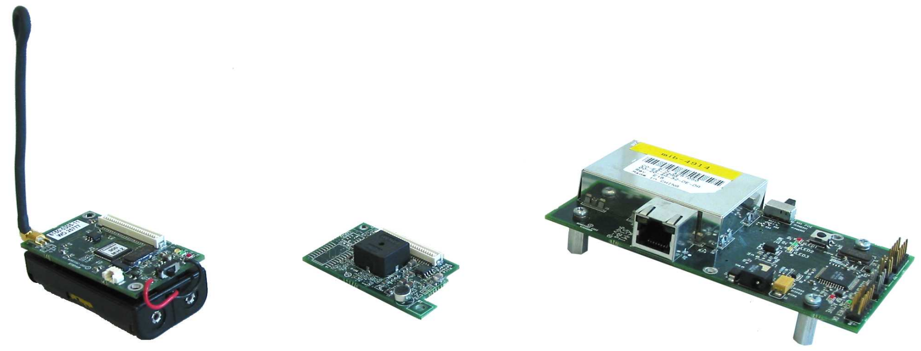


Figure 1.2: A mica2 sensor (left), a MTS300CA sensor board for measuring light, temperature and sound (center), and a MIB600 ethernet gateway which is used for programming the nodes and connecting them to a network (right).

for embedded networked sensors. Since it is released under an open-source license, it is available for free and it was ported to other embedded platforms. The TinyOS software package comes with the most important library modules (network stack, hardware drivers, ...) and the tools which allow the mote to communicate with a Java application running on a computer.

Due to the very limited resources available on most embedded sensor devices, TinyOS has some heavy constraints:

- There is no concurrency in TinyOS, i.e. only one task (this is how threads are called in TinyOS) can be running at the same time. This task can not be interrupted by another task. The only exception of this rule are hardware interrupts which may interrupt the current task at any time[3]. Nevertheless, it is possible to handle multiple activities in parallel: The current task can create new tasks which are placed into the TinyOS task queue. This queue is processed in FIFO[4] order when the current task finishes. The advantage of the restriction of only one concurrent task is that the compiler can detect data races[5] at compile time. The disadvantage is that the program structure becomes more complicated because bigger jobs need to be split up in multiple small tasks, disguising their correlation. This makes TinyOS applications harder to write and difficult to understand.

- TinyOS does not support dynamic memory allocation. Everything is allocated on the stack, there is no heap in TinyOS. This restriction greatly influences the design of TinyOS applications: Data structures must be initialized to their maximum size at compile time and therefore probably reserve too much of the limited memory of small sensor nodes. However, due to this restriction, a TinyOS application may hardly contain memory leaks.

[3]Except when using the *atomic* keyword, see [5].

[4]First in, first out

[5]Data races may occur when a variable is accessed by a hardware interrupt handler and by a task of the module simultaneously.

- The size of radio messages sent using the TinyOS library is limited to 29 bytes of payload. This implies that bigger data packets must be split into smaller parts. Splitting, transmitting (including possible retransmission of lost or corrupted packages) and recombining has to be done manually, there is no library offering this service.

The programming language of TinyOS is a descendant of C called nesC [5]. It uses the C syntax and adds some new constructs. The most important concept of nesC is the separation of construction (the *implementation*) and composition (the *wiring*): A nesC application consists of at least one component (also called module) and one configuration file. The modules contain the actual implementation while the configuration specifies how the modules are wired together. Communication between modules is done using interfaces. These interfaces are — opposed to interfaces in popular languages like Java — bidirectional. For example, the interface of a timer may offer commands to start and stop the timer, but it forces the user of the interface to supply an implementation for an event executed when the timer fires.

Compilation of nesC applications is done in two steps: In the first step, the nesC compiler takes all required components and generates one big C file. This file is then used by the gcc compiler to generate the mica2 binary in a second step. The whole compilation process is transparent to the user: A makefile included by the default installation of TinyOS handles this process.

1.4 Assignment Description

The task of this diploma thesis is to implement the XTC algorithm on the mica2 motes. This includes the following subtasks:

- Getting familiar with TinyOS and nesC. During this preparation phase, small applications were created to understand and test the relevant properties of the mica2 motes.

- Understanding the XTC algorithm described in [9]. The algorithm has to be extended to work correctly in a dynamic environment.

- Developing reasonable link quality metrics with respect to the limited hardware resources of the mica2 motes.

- Implementation of the XTC algorithm using the nesC programming language and a Java application to display the results.

- Devising a sample application that highlights the benefit when it is run on the topology established by XTC compared to running it directly on the initial network.

- Testing and performance evaluation of the implementation in a real environment.

Chapter 2

Preliminary Work

To become acquainted with the nesC language and the different concepts of TinyOS, several small applications were created. Note that these applications are not directly connected to the XTC algorithm, however they might very well be used for educational purposes for people starting to work with TinyOS since the applications show in few lines of code how to use some specific features of nesC, TinyOS, and the mica2 motes. Additionally, programs like the EEPROM Editor might turn out to be a useful tool when working with the mica2 sensors. In the following, we will discuss the application individually in more detail.

2.1 Remote Control Application

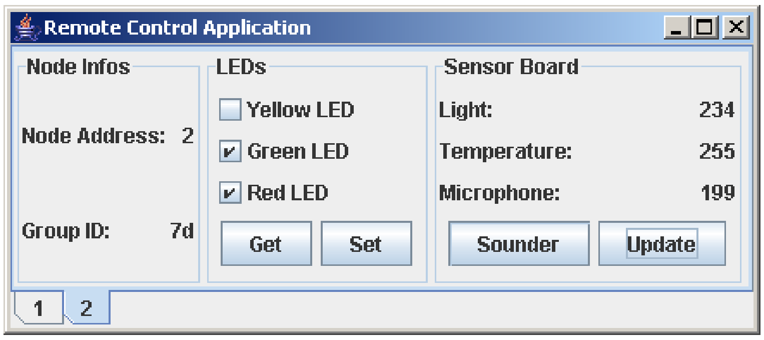


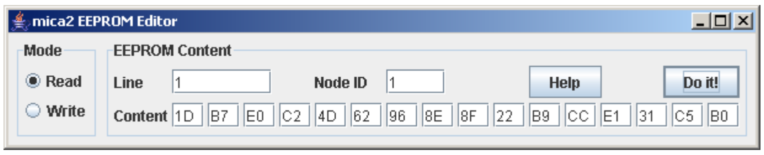
Figure 2.1: The Remote Control Application: Currently, two nodes with IDs 0x01 and 0x02 are found.

This application allows to read and change the state of a node, i.e. the state of the LEDs, the data of the sensors (if a sensor board is connected), and the state of the beeper (on/off) on the sensor board. For each discovered node, a new tab is added (see Figure 2.1).

The aim of this application was to familiarize with the data exchange between a Java application and the mica2 motes. This includes learning to work with tools the TinyOS software package provided for this task: The mig tool to create Java classes out of a message declaration in a nesC header file, the

SerialForwarder to relay messages from a mica2 sensor through the ethernet gateway to a Java application, and the `net.tinyos` package to send and receive messages within a Java application.

2.2 EEPROM Editor



Line: 1
Node ID: 1
Content: 1D B7 E0 C2 4D 62 96 8E 8F 22 B9 CC E1 31 C5 B0

Figure 2.2: The mica2 EEPROM Editor

The EEPROM Editor provides the functionality to read from and write to the EEPROM of the mica2 motes. The EEPROM is used by the LoggerWrite and LoggerRead modules to store data in a persistent manner, i.e. the data is available even after a reboot of the node. The EEPROM consists of 2^{15} lines (0 to 32767), each of size 16 bytes. The first 16 lines are reserved for the system, that is, they cannot be accessed by using the LoggerRead and LoggerWrite interfaces. The application therefore uses the lower level interfaces EEPROMRead and EEPROMWrite which allow reading and editing all lines.

The first 16 lines are used by components like Deluge [4] to store internal information. Because of that, the EEPROM Editor might prove to be a useful tool when analyzing such applications.

2.3 Topology Displayer

This tool draws the network topology. Unlike comparable tool like Surge or the topology display features of TinyDb (Surge and TinyDb are part of the TinyOS software package), the complete topology is drawn and not only a minimum spanning tree built on top of the network graph. The Topology Displayer can be used either as stand-alone application or as a module as part of another application.

The Java front-end of this application is shown in Figure 2.3. It formes the basis of the *XTC Interface* described in Chapter 3.5.

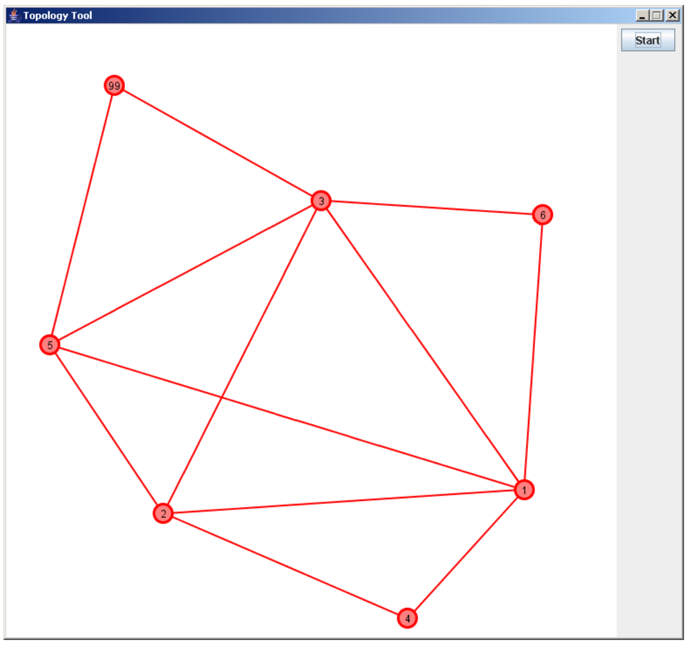


Figure 2.3: The Topology Displayer application showing the network topology of 6 nodes (IDs 1 to 6) including the gateway node with ID 99 used to inject messages into the sensor network.

Chapter 3

Implementing XTC

3.1 Static vs. Dynamic XTC

The XTC algorithm as described in [9] does not handle dynamic changes of the network graph: The topology graph G_{xtc} is only calculated once. When the link quality changes, G_{xtc} is not updated. If a link is disconnected or a node is removed from the network graph (e.g. when the batteries are empty or by simply turning it off), G_{xtc} may even become disconnected. Additionally, it is not possible to add new nodes once the algorithm is executed.

However, changes to the link quality and adding or removing nodes is very common in a real environment. For this reason, the XTC algorithm had to be extended to correctly handle such events. The requirement of the dynamic XTC algorithm was — besides producing correct topology graphs with the updated data — to prevent global changes in G_{xtc} as a reaction to a local modification.

The adapted dynamic version of XTC is basically a loop with the original XTC algorithm inside: Instead of calculating G_{xtc} once, it is (re)calculated in every iteration of the loop. Every iteration is called update cycle. In each update cycle, the node searches for new neighbors and checks if its known neighbors are still alive. Then it updates its link quality values and recalculates G_{xtc}. That way, adding and removing nodes and changes in the link quality are handled correctly. But there is one more piece of information required to calculate G_{xtc} which might have changed: the order of the neighbors of a node. This is handled the following way: When a node detects that its own order has changed, it informs its neighbors about that fact. In doing so, each node already knows during its update cycle which orders it has to update. It can therefore simply poll the corresponding neighbors for their updated order.

Using this approach, local changes of the network cannot lead to global changes in G_{xtc}:

- A change of the link quality is only recognized by the two endpoints of the link. The new value is not directly propagated to any other node.

- A node recently added to the network graph G can only be seen by its direct neighbors. The same is true for the failure of a node. Like above, this information does not directly spread out in the network.

- The information above is only indirectly propagated by a message that the order has changed. Since this knowledge does not affect the own order, this information is not transmitted further.

3.2 Architecture

The implementation of the XTC algorithm consists of three main parts: A data structure that manages the links and their calculated quality values at each node (the private order of a node), a second data structure to keep track of the orders of the neighbors, and the actual program logic which contains the algorithm to calculate the resulting topology G_{xtc}. These three parts are capsuled in three modules, namely PrivateOrder, NeighborOrder, and XtcNode (the program logic). The next sections give a detailed overview of each of these modules.

3.2.1 The PrivateOrder Module

The PrivateOrder module consists of the files PrivateOrderM.nc (the implementation), PrivateOrder.nc (the corresponding configuration), and POrder.nc (the interface provided by PrivateOrderM.nc). The task of the POrder module is to collect information about all links of a node. This information includes:

- The most recent quality value of each link including a time stamp containing the time when the link quality was evaluated the last time,
- a flag displaying whether the link is currently part of the G_{xtc}, and
- a flag which is set if the link was already processed in the current algorithm iteration.

The links are ordered according to their quality, starting with the lowest (i.e. best) value. Besides this link management, the PrivateOrder module offers a command called `getOrder` which writes the ordered list of the neighbor IDs to a buffer. This command is used by the XtcNode module when exchanging the order with its neighbors.

3.2.2 The NeighborOrder Module

The NeighborOrder module consists of the files NeighborOrderM.nc, NeighborOrder.nc and NOrder.nc with similar semantics as the files of the PrivateOrder module. It is used to manage the orders of all direct neighbors of a node.

Due to the fact that the size of radio messages is limited to 29 bytes of payload data, the order of nodes with more than 12 neighbors[1] must be split into multiple parts for transmission. The reconstruction of an order from is different parts is also done inside the NeighborOrder module.

[1]29 bytes payload minus 4 bytes message overhead (2 bytes containing the sender ID, 1 byte specifying the part number, 1 byte for the total number of neighbors which is required to calculate the total number of parts) leaves 25 bytes for the actual order data. Since nodes IDs require 2 bytes, this leaves room for 12 node IDs.

Besides the management of the orders, the NeighborOrder module offers the method `compare` to compare two node IDs according to their quality using the order of a neighbor. The result of this method forms the basis for the decision of the XTC algorithm, that is, whether or not to include a link in G_{xtc}.

3.2.3 The XtcNode Module

The XtcNode module consists of the files XtcNodeM.nc and XtcNode.nc, i.e. the module and its configuration. Additionally, there are two header files: XtcMsg.h specifies the messages that are sent by the algorithm (see Chapter 3.3) and Xtc.h containing some global constants used by all modules.

Furthermore, this module contains the program logic. This includes the handling of incoming messages, generation of output messages, and the actual XTC algorithm, i.e. the decision which links should be kept active. It uses the POrder and NOrder interfaces provided by the corresponding modules to keep track of its neighbors and their orders. A detailed description how this module works is given in Chapter 3.4.

3.3 Message Types

The XTC algorithm uses five different message types to calculate G_{xtc}. These messages are — together with the two message types used by the *XTC Interface* (see Chapter 3.5) — defined in the XtcMsg.h file. This section describes the function of these messages and how a node reacts when receiving them.

Some message types contain a field called action. This field is used to specify the subtype of the message, e.g. request or acknowledgement. The decision was made to use this approach instead of defining a special message type for each possible action because a topology control algorithm is most probably only a small part of a bigger application. Occupying too many message types (only 256 are available in total) could unnecessarily limit the number of message types for the rest of the application. The disadvantage of using an additional field to specify the subtype is that the payload size of such messages is reduced by one byte.

3.3.1 NbSearchMsg Message

This message is sent at the beginning of each update cycle. It is used to search for new neighbors. The action field of a `NbSearchMsg` message is either set to `SEARCH_ACTION_PING` or `SEARCH_ACTION_PONG`. The first setting is used to start a search for new neighbors while the latter is utilized by the new neighbors to inform a node about their presence.

Figure 3.1 shows the process of searching for new neighbors: Node 1 broadcasts a `NbSearchMsg` message to search for new nodes in its neighborhood. It appends the list of all its currently known neighbors (node 3 and 7 in this example) to this message and sets the action field to `SEARCH_ACTION_PING`. Its neighbors 3, 7, and 9 receive this message. All of them check whether their ID is already in the list of known neighbors. Since node 9 does not find its ID in the list, it replies with a `NbSearchMsg` message with the action field set to `SEARCH_ACTION_PONG`. Nodes 1 and 9 are now aware of the new link between

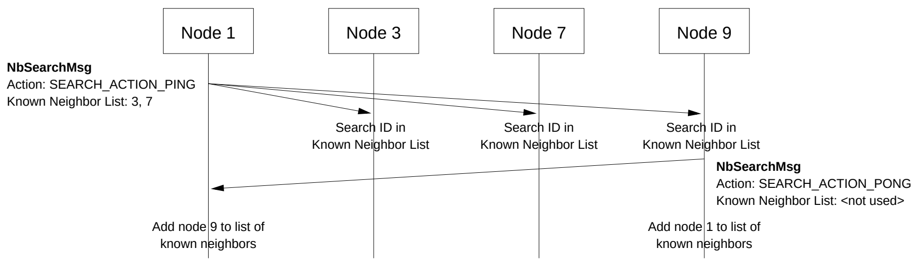

Node 1
Node 3
Node 7
Node 9
NbSearchMsg
Action: SEARCH_ACTION_PING
Known Neighbor List: 3, 7
Search ID in Known Neighbor List
Search ID in Known Neighbor List
Search ID in Known Neighbor List
NbSearchMsg
Action: SEARCH_ACTION_PONG
Known Neighbor List: <not used>
Add node 9 to list of known neighbors
Add node 1 to list of known neighbors

Figure 3.1: Node 1 searching for new neighbors using `NbSearchMsg` messages.

them. Because node 3 and 7 find their IDs in the list of known neighbors, they just ignore this message.

3.3.2 OrdCalcMsg Message

To calculate the quality of a link, `OrdCalcMsg` messages are used. The action field of this message type can be set to three values: `ORD_CALC_REQ`, `ORD_CALC_-INTER`, or `ORD_CALC_ACK`.

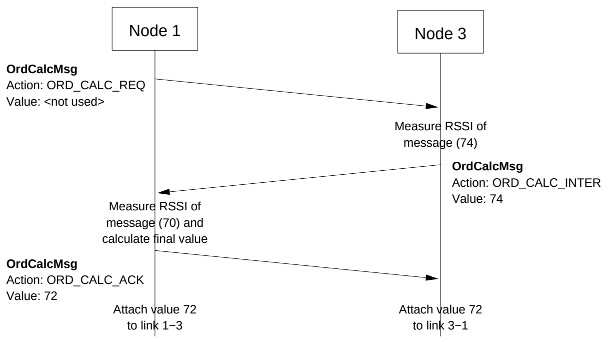


Figure 3.2: Link quality calculation between the nodes 1 and 3.

The process of calculating the link quality is shown in Figure 3.2: Node 1 initiates the calculation by sending an `OrdCalcMsg` with the action field set to `ORD_CALC_REQ`. Node 3 measures the RSSI[2] value upon receipt and writes this value to the answers with action set to `ORD_CALC_INTER`. When node 1 receives this message, it measures its RSSI, too. Given the two RSSI values, it calculates the final link quality value (see Chapter 3.3 for details). This value is sent with a final `OrdCalcMsg` message (action=`ORD_CALC_ACK`) back to node 3. Both nodes save this new value.

[2]**R**eceived **S**ignal **S**trength **I**ndicator: This value is attached to each received packet by the TinyOS network stack. It contains information about the signal strength of the packet. The smaller the value, the stronger was the received signal.

3.3.3 OrdReqMsg and OrdSendMsg Messages

The `OrdReqMsg` and `OrdSendMsg` messages are used to exchange the orders between neighbors. They both do not contain an action field. A node u sends the request for a part of the order[3] of its neighbor v using an `OrdReqMsg`. Node v answers to this request with an `OrdSendMsg`. This message contains — besides the demanded part of the order — the part number and the total number of neighbors of the node. This information is required to correctly reassemble the parts at node u.

3.3.4 OrdChangedMsg Message

Messages of type `OrdChangedMsg` are broadcasted by nodes after their order has changed. This informs all neighbors that they need to update the order of the sender in their next update cycle. Therefore, all nodes delete the corresponding order in the NeighborOrder module upon receipt an `OrdChangedMsg` message.

3.4 Functionality of the XtcNode Module

3.4.1 State Diagram

Figure 3.3 shows the state diagram of the XtcNode module. The different states are described in the following. For the sake of clarity, the diagram does not show all details and the reactions on most incoming messages — a complete list of how to react on messages is given in Chapter 3.3.

Basically, the state diagram consists of three steps, each representing one step of the XTC algorithm:

1. Updating the PrivateOrder module. This includes searching for new neighbors and reevaluation the quality of existing links if the current value is not up-to-date anymore (i.e. the time stamp is too old).

   Corresponding states:
   `STATE_SEARCH_NEIGHBORS`, `STATE_UPDATE_PORD`, and `STATE_PARALLEL_WAIT`.

2. Updating the NeighborOrder module. In this step, the order of newly found neighbors is requested and collected in the data structure. Additionally, the order of an old neighbor might need to be updated if it was changed. Since a node has to inform its neighbors if its order changes, there is no need to poll all nodes for an update of their orders.

   Corresponding states:
   `STATE_UPDATE_NORD` and `STATE_AWAIT_NORD_ACK`.

3. Recalculate G_{xtc} with the new data. This step may be skipped if neither the PrivateOrder nor the NeighborOrder module has changed.

   Corresponding state:
   `STATE_CALC_TOPOLOGY`

In the following, we describe all these states in more details.

[3]Orders might be split in multiple parts, see Chapter 3.2.2.

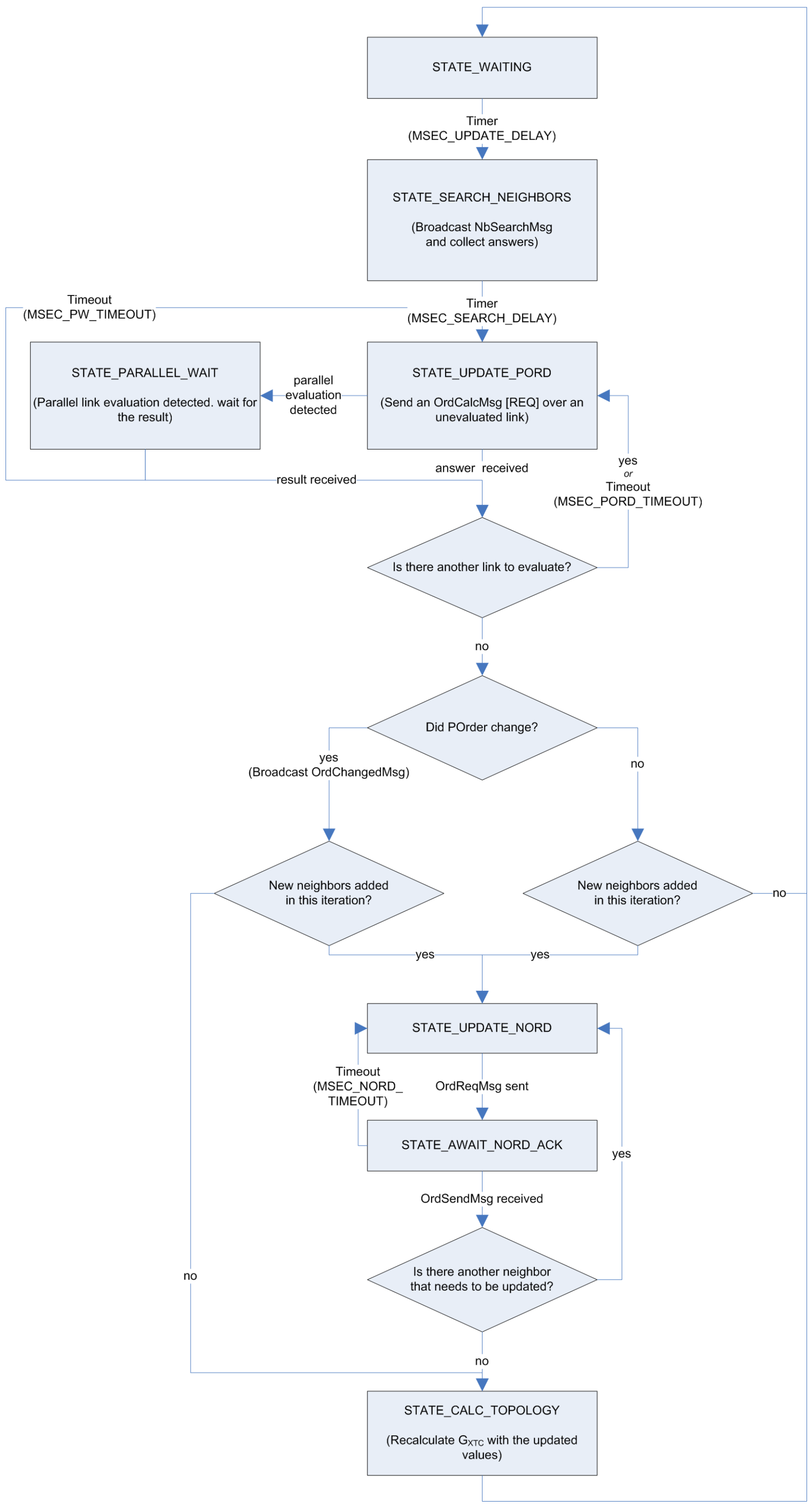

STATE_WAITING
Timer (MSEC_UPDATE_DELAY)
STATE_SEARCH_NEIGHBORS
(Broadcast NbSearchMsg and collect answers)
Timeout (MSEC_PW_TIMEOUT)
Timer (MSEC_SEARCH_DELAY)
STATE_PARALLEL_WAIT
(Parallel link evaluation detected. wait for the result)
parallel evaluation detected
STATE_UPDATE_PORD
(Send an OrdCalcMsg [REQ] over an unevaluated link)
answer received
result received
yes or Timeout (MSEC_PORD_TIMEOUT)
Is there another link to evaluate?
no
Did POrder change?
yes (Broadcast OrdChangedMsg)
no
New neighbors added in this iteration?
New neighbors added in this iteration?
no
yes
yes
STATE_UPDATE_NORD
Timeout (MSEC_NORD_ TIMEOUT)
OrdReqMsg sent
STATE_AWAIT_NORD_ACK
yes
OrdSendMsg received
Is there another neighbor that needs to be updated?
no
no
STATE_CALC_TOPOLOGY
(Recalculate G_XTC with the updated values)

Figure 3.3: The state diagram of the XtcNode module

STATE_WAITING

The XtcNode module switches to this state when an algorithm iteration is finished. The module is waiting for a `Timer.fired()` event to start the next iteration. Even though there is no action initiated before the timer fires, incoming messages are processed and an answer is generated, if appropriate.

STATE_SEARCH_NEIGHBORS

This state represents the first step in each algorithm iteration. Node u broadcasts a `NbSearchMsg` message (action = `SEARCH_ACTION_PING`) containing a list of all known neighbors. If another node v receives such a message and does not find its own node ID in the list, it answers with a `NbSearchMsg` message with the action field set to `SEARCH_ACTION_PONG`. This informs u that v is a new neighbor (and vice versa). Node u then waits a certain period of time (defined in `MSEC_SEARCH_DELAY`) for new neighbors to answer. After this delay, it proceeds to the next state and new neighbors are rejected until the next iteration of the algorithm.

STATE_UPDATE_PORD and STATE_PARALLEL_WAIT

As the name suggests, the goal of the `STATE_UPDATE_PORD` state is to update the link quality values in the PrivateOrder module. This is done by iterating through all known neighbors. If the value is considered up-to-date[4], the current link is skipped. Otherwise, an `OrdCalcMsg` message is sent to the neighbor to start the link quality evaluation. This process is described in Section 3.3.2. If a node fails to react `ORD_CALC_RETRIES` times to such a message, this node is considered out of reach and is removed from the PrivateOrder and NeighborOrder modules.

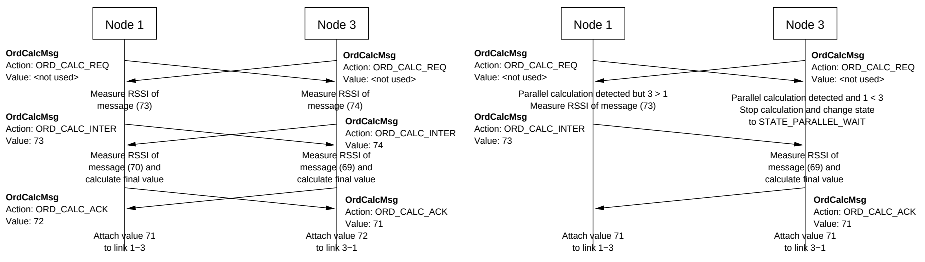


Figure 3.4: Both nodes decide to evaluate a link at the same time. On the left, this results in two different values. This erroneous scenario is resolved by the `STATE_PARALLEL_WAIT` state. The scenario at the right shows the correct process when both nodes start the evaluation of the same link at the same time: The request of the node with the smaller ID is ignored and only the other request leads to a new link quality value.

The STATE_PARALLEL_WAIT state is used only to prevent inconsistencies

[4]This happens about 50% of the time: For two connected nodes u and v, one of them (say u) starts the quality evaluation. The new value is saved by both nodes with the current time as time stamp. Node v — most probably in `STATE_WAITING` — will skip the evaluation of this link in its next update cycle because the time stamp is still up-to-date, i.e. (current time - time stamp) < `UPDATE_INTERVAL`.

in one special case: When both endpoints of a link decide to evaluate the link quality exactly at the same time, the link calculation protocol would run twice in parallel resulting in two (possibly slightly different) values saved at each node (see Figure 3.4 in the middle). This difference could result in a topology graph which is not connected, since multiple links might get multiple values (see Theorem 4.1 in [9]).

This special case is prevented as shown in the scenario on the right in Figure 3.4: If a node detects such a parallel link evaluation (i.e. if it receives an initializing `OrdCalcMsg` message from a node to which it just sent such a message itself), it stops the calculation and switches its state to `STATE_PARALLEL_WAIT` if its own node ID is smaller than the ID of the other node. Since both nodes detect the parallel evaluation, it is guaranteed that one of them stops.

While one might think that this scenario will occur very rarely, in practice it happens quite often. A very simple scenario illustrates one possibility of a parallel link evaluation: Three nodes u, v, and w, each of them connected with the other two nodes. Nodes u and v are in state `STATE_WAITING` and w is performing an update cycle. Because the order of node w changes due to a new link value, it sends an `OrdChangedMsg` message. This message causes its two neighbors to start their update cycle. Since the links u-w and v-w are up to date (node w just updated them), node u and v start updating the link u-v at the same time.

STATE_UPDATE_NORD and STATE_AWAIT_NORD_ACK

Like the `STATE_UPDATE_PORD` state, the task of the `STATE_UPDATE_NORD` state is to update the data structures required by the XtcNode module. Therefore, a loop iterates over all neighbors. If the order of a neighbor is not available in the NeighborOrder module a request is sent to the corresponding node. Since the order might be too big for one `OrdSendMsg` message, it is possible that multiple requests for different parts are required until the order of one neighbor is complete.

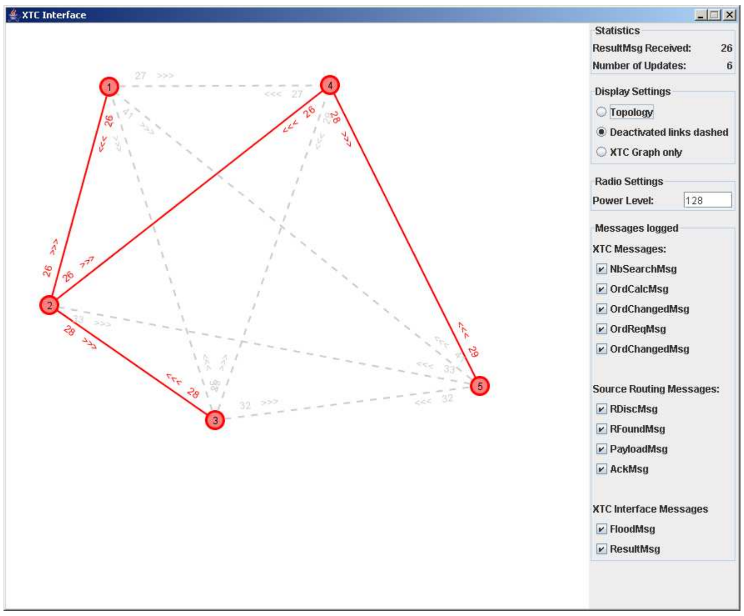
When the request is sent, the node changes to the `STATE_AWAIT_NORD_ACK` state. It switches back to state `STATE_UPDATE_NORD` either when a timeout occurs or when the requested part of the order is received.

STATE_CALC_TOPOLOGY

This is the last step of the algorithm where the XtcNode module (re)calculates G_{xtc} using the most up-to-date data. After the calculation, the node switches back to `STATE_WAITING` state.

3.5 XTC Interface

Small sensor nodes like the mica2 motes have a big drawback: There is no possibility to display any information directly on the nodes except turning on and off the three LEDs. For displaying and studying the results of the XTC algorithm (and also for debugging during development), three LEDs are not sufficient. Therefore, a special application had to be created. This application called *XTC Interface* is written in Java since TinyOS provides tools like mig[5] and the SerialForwarder[6] and because Java offers best compatibility to multiple operating systems.



Statistics
ResultMsg Received: 26
Number of Updates: 6

Display Settings
○ Topology
● Deactivated links dashed
○ XTC Graph only

Radio Settings
Power Level: 128

Messages logged
XTC Messages:
✓ NbSearchMsg
✓ OrdCalcMsg
✓ OrdChangedMsg
✓ OrdReqMsg
✓ OrdChangedMsg

Source Routing Messages:
✓ RDiscMsg
✓ RFoundMsg
✓ PayloadMsg
✓ AckMsg

XTC Interface Messages
✓ FloodMsg
✓ ResultMsg

Figure 3.5: The XTC Interface application displaying the result of the XTC algorithm with five nodes running. The topology graph is drawn in red, the dashed gray lines are deactivated by XTC.

The user interface of *XTC Interface* is based on the interface of SANS [1], a network simulator for Java applications. It allows moving the nodes using drag and drop. That way, the nodes can be placed equivalent to their position in the real world to allow tracing the XTC algorithm.

XTC Interface automatically searches for nodes running XTC and collects data from them. This is done with two new message types:

- `FLOOD_MSG` messages used to establish the routes for further data acquisition are broadcasted by the *XTC Interface*. They start a data collection iteration. Each node receiving such a message rebroadcasts it exactly once per iteration. Additionally, the node saves the source of the first `FLOOD_MSG` message in the current iteration. In doing so, the `FLOOD_MSG` messages are used to build a communication tree with the *XTC Interface* at its root (or more precisely, the gateway to which *XTC Interface* is connected). This tree is then used to collect the data. It is rebuilt for each data collection iteration.

- Additionally to forward the `FLOOD_MSG` messages, each node sends — after a short random delay to prevent a broadcast storm [8] — a `RESULT_MSG`

[5]mig is used to create Java classes out of a message declaration in a nesC header file.

[6]Ethernet gateways like the MIB600 inject packets sent from the connected node into a network. The SerialForwarder reads such packets and acts as packet distributor for other applications.

message to the source of the first FLOOD_MSG message received in this iteration. This message contains the node's current state of the XTC algorithm: A list of all neighbors and the corresponding link quality values of them. Moreover, a flag declaring whether or not this link is part of G_{xtc} is included. When a node receives such a RESULT_MSG message, it forwards the message to the source of the first FLOOD_MSG message. In the end, all RESULT_MSG messages are routed on the tree graph to the *XTC Interface*.

With the collected RESULT_MSG messages, *XTC Interface* draws the current state of the complete sensor network. Depending on the settings, either the whole topology, G_{xtc} with inactive links dashed, or only G_{xtc} is drawn. If no RESULT_MSG message is received from a node in the data collection iteration, the data from the last iteration is used. Since this data might be out of date, old data is drawn with lighter shade of red in *XTC Interface*. With each iteration, the shade gets nearer to white. This informs the user that this particular data is out of date and might be incorrect. After five iterations without receiving a RESULT_MSG message, a node is considered disconnected and is therefore displayed as a "ghost"-node without any links.

Even if a RESULT_MSG message is received from every node there may be some inconsistencies in the topology graph (e.g. a link with two different quality values or even a cycle of length three in the topology graph[7]). This is neither due to bugs in the implementation nor a general problem of XTC but because the nodes may send the RESULT_MSG message during their update cycle where temporary inconsistencies are possible[8]. Such errors disappear in the next data collection iteration.

The *XTC Interface* writes a log file containing the messages received. By default, all messages are logged but this can be changed by deselecting the checkboxes corresponding to the different message types used by XTC, the Dynamic Source Routing presented in the next chapter, and the *XTC Interface* itself. The log file called *logfile.txt* is created in the same directory as the application. At each startup, the log file is cleared by removing old entries.

3.6 Link Quality Metrics

The XTC algorithm is based on a very abstract concept of link quality. It is up to the implementation to choose what properties of a link are measured and what metrics are used.

The earliest working version simply used the RSSI value of a message sent over a link. There was no negotiation between two neighbors about a common value but each node saved its own value. This resulted in topology graphs that were not connected. The reason for this lies in the assumption of theorem 4.1 in [9]: Connectivity is only guaranteed on Euclidean Graphs. Therefore, the agreement protocol described in Section 3.4.1 had to be implemented.

[7]Cycles of length three are not possible in the topology graph when using symmetric links (see Theorem 5.2 in [9])

[8]Delaying the RESULT_MSG message until the update cycle is complete does not resolve this problem: Since the update cycles of the nodes do not occur concurrently, the RESULT_MSG messages received would not represent a common point in time and could therefore contain inconsistencies, too.

The current version of the algorithm supports two different metrics. Both of them are based on the RSSI values of packets sent between two nodes. In both metrics, the node collects two RSSI values, one from its neighbor and one from itself. In the first metrics, these two values are averaged. This makes sense if the assumption is made that the difference between the values arises from the fact that the radio modules at different nodes are not calibrated equally. The second metric available in the implementation takes the maximum of the two values. This approach presumes that the two values differ due to different settings of the neighbors, e.g. different radio power settings.

There are many more possible metrics. However, due to time constraints only the two metrics mentioned above were implemented. Some additional ideas for new metrics are noted in Section 6.2.

Chapter 4

Dynamic Source Routing

To illustrate the benefit of the XTC algorithm, a sample application using the XTC graph G_{xtc} as underlying network topology was implemented. The decision was made to realize a dynamic source routing (DSR) algorithm which is described in this chapter. The comparison between running DSR on G_{xtc} versus running it on the complete network graph G is given in Section 5.2.

4.1 Algorithm Description

The basic principle of the dynamic source routing algorithm is the following: If a node u wants to send a message to a node v, it appends the route between u and v to the message. All nodes on this path forward the message according to the given route. This is called the *forwarding phase*.

There are multiple possibilities for node u to find a way to v (the *route discovery phase*). The implementation realized in this thesis uses a controlled flooding: Node u broadcasts a route discovery message containing the node ID of the destination and a TTL[1]. Each node receiving such a message forwards it once. Before forwarding the message, nodes attach their own ID to the route discovery message and decrement the TTL field. Once the TTL reaches 0, the message is not forwarded anymore. When the destination v receives such a route discovery message, the message contains the path between u and v because each node that forwarded the message attached its ID. Node v now replies with a route found message containing the complete path from u to v.

The TTL is used to prevent the flooding of the whole network even though the destination might be very close to the source of the message. With a small TTL only short paths are searched. If no route is found (i.e. no route found message is received at the source after a certain amount of time) the source increases the TTL and restarts the route discovery.

There are countless possible tweaks to increase the performance of dynamic source routing. Approaches include caching of routes for a certain amount of time (or until the route fails), using local search to repair broken routes and multiple variations of how acknowledgement messages are used (or even use an implicit acknowledgement). However, only the basic algorithm described above is implemented as part of this thesis.

[1]Time to live

4.2 Implementation

The DSR implementation uses four types of messages:

- **RDiscMsg** messages are used during the route discovery phase. Besides the source and the destination of the inquired route, this message contains — as mentioned in Section 4.1 — a TTL field. Because each node only forwards such a `RDiscMsg` message at most once, it must also contain a unique identifier. The identifier consists of the ID of the node at the start of the route together with a counter which is incremented each time this node starts a new route discovery. That way, a node only forwards a `RDiscMsg` message if its counter is bigger than the counter of the last forwarded message with the same route source. The last field of `RDiscMsg` messages contains a list with the current route of the message. Each node that forwards the message appends its own ID at the end of this list.

- **RFoundMsg** messages are sent to the source of a route discovery when a `RDiscMsg` message arrives at the destination of the route. It contains the same fields as the `RDiscMsg` message, except for the missing TTL. On recept of a `RFoundMsg` message, it forwards the message according to the route contained in the message — there is no need to broadcast `RFoundMsg` messages.

- **PayloadMsg** messages contain the payload data the source wants to send to the destination. Besides the data, it contains the route which was discovered earlier. It also contains the counter used in the `RDiscMsg` and `RFoundMsg` messages. The counter might seem unnecessary, but it is later required for the `AckMsg` message (see below) to specify which transaction was successfully completed.

- **AckMsg** messages are sent from the route destination to the source after the `PayloadMsg` message was received. Since it is possible that a route breaks after the `RFoundMsg` message was sent back to the source, a `PayloadMsg` may get lost. Therefore, the source will restart the route discovery phase if no `AckMsg` message is received after a certain time. The `AckMsg` message contains the same fields as the `RFoundMsg` message: source and destination including the route in-between and the counter. This implementation additionally includes two fields called *rssiAvg* and *rssiMax*. These fields are not used for the DSR algorithm but for collecting data as described in Section 4.3.

Due to the limited size of 29 bytes per packet, two restrictions were made in the implementation: Only node IDs up to 255 are supported and the maximum route length is 8 hops (not including the source and destination). Theoretically, every node ID is 16 bits wide. However since a `PayloadMsg` message must contain the whole route and additionally leave some space for the actual payload data, only 8 bits are used per node on the route. With a maximum route length set to 8 (declared in the `MAX_ROUTE_LENGTH` constant in the file SourceRoutingMsg.h), this allows only 16 bytes of real payload data. Longer routes or more bits per node ID would reduce this value even more.

4.3 Data Logging

The DSR algorithm was implemented to illustrate the effect of using XTC and topology control in general. Therefore, logging of data is very important. The acquired data is written to the EEPROM. That way, the sensor nodes can be turned off without losing data. Turning off the nodes may be required since the data collection protocol does not support multi-hop routing: The PC queries (through a gateway node, see Section 5.2.1) one node after the other for all of its logger entries. If a node is out of reach of the gateway node, no data is received.

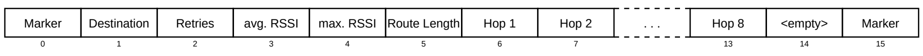


Figure 4.1: The structure of one EEPROM line representing one logger entry, i.e. one route.

The structure of the collected data inside the EEPROM is straightforward: Each line (16 bytes) corresponds to one route. Figure 4.1 shows what information is stored inside one page. The first and the last byte of each page contain a marker. These bytes mark the corresponding page as a legal logger entry. Byte one contains the destination of the route — there is no need to save the source of the route, since the source corresponds to the node writing the logger entry. Byte two represents the number of retries before the process was successfully completed, i.e. the route was found, a payload packet was transmitted, and an `AckMsg` message was received by the source. The average RSSI value on the route is written to byte three. This value is accumulated inside the data field of `PayloadMsg` messages. Since there is no real payload sent during this experiment, this action is completely acceptable. In the `AckMsg` message, the average and maximum RSSI values are sent back to the source. The remaining bytes of the EEPROM page contain the route: Byte four specifies the length of the route, bytes five to thirteen the IDs of the nodes between source and destination. Byte fourteen is not used.

The Java tool which collects the logger entries of all nodes produces output like the following:

```
...
Message from 1 to 4:
Route:  1 -> 4 (length=0)
Number of retries:  0
Average RSSI: 24
Maximum RSSI: 24

Message from 1 to 3:
Route:  1 -> 4 -> 3 (length=1)
Number of retries:  1
Average RSSI: 28
Maximum RSSI: 32

Final statistics for node 1:
Total routes:  104
```

```
Average route length:  1.83
Maximum route length:  5
Average RSSI: 27
Maximum RSSI: 45
...
```

The information contained in this output is compiled into the table shown in Chapter 5.2.

Chapter 5

Performance Evaluation

5.1 XTC in a Building

To evaluate the performance of the implementation of the XTC algorithm, it was tested in a real environment. The setup of this experiment and the results including their interpretation are presented in this chapter. The detailed pictures of this test can be found in Appendix A.

5.1.1 Test Setup

The twelve nodes were placed on the floor of an office building according to the map shown in Figure A.1 in the appendix. Those nodes form the network graph G. At the border of G — in the upper right corner of the map — a gateway node was added. This node connects the sensor network to the *XTC Interface* which was used to display the results. The gateway node does not appear in the *XTC Interface*. It is only used to inject packets into the network and to read messages from the network.

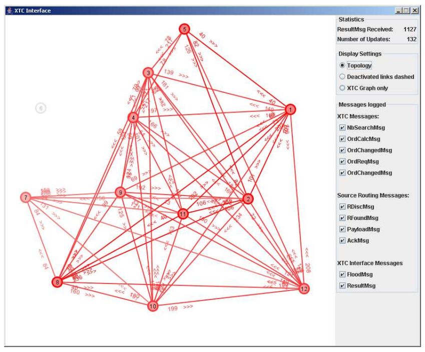
All nodes were sending at a power level set to 128. As link quality metrics, the average between the two measured RSSI values was used (see Section 3.6).

5.1.2 Results

During the experiment, node 6 showed strange behavior. This was probably due to low batteries which resulted in a high packet loss and therefore outdated data at this node. Because of that, it was turned off and appears greyed out in the picture. The other eleven nodes did not show such a behavior.

The left image in Figure 5.1 depicts the network topology G of the twelve nodes. At the right, the topology graph G_{xtc} produced by the XTC algorithm is displayed. As it can be seen, XTC was able to drastically reduce the number of links in G while preserving connectivity. However, when comparing G_{xtc} to the map shown in Figure A.1, one might expect that long links like the one between node 1 and node 8 should be removed. But since the measured RSSI value for this link was very low (i.e. the message was received with a high power level), this link was included in G_{xtc}.

The result of this experiment is — besides the awareness that the implementation works correctly — that the RSSI basis for the link quality metric is



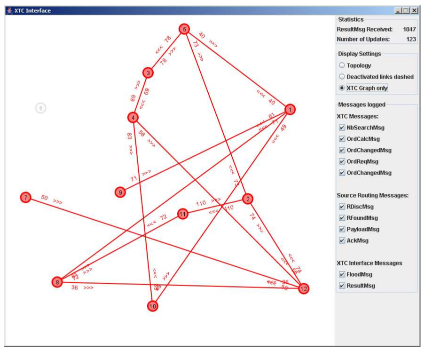
Statistics
ResultMsg Received: 1127
Number of Updates: 132

Display Settings
[✓ Topology]
Deactivated links dashed
XTC Graph only

Messages logged
XTC Messages:
✓ NbSearchMsg
✓ OrdCalcMsg
✓ OrdChangedMsg
✓ OrdReqMsg
✓ OrdChangedMsg

Source Routing Messages:
✓ RDiscMsg
✓ RFoundMsg
✓ PayloadMsg
✓ AckMsg

XTC Interface Messages
✓ FloodMsg
✓ ResultMsg



Statistics
ResultMsg Received: 1047
Number of Updates: 123

Display Settings
Topology
Deactivated links dashed
[✓ XTC Graph only]

Messages logged
XTC Messages:
✓ NbSearchMsg
✓ OrdCalcMsg
✓ OrdChangedMsg
✓ OrdReqMsg
✓ OrdChangedMsg

Source Routing Messages:
✓ RDiscMsg
✓ RFoundMsg
✓ PayloadMsg
✓ AckMsg

XTC Interface Messages
✓ FloodMsg
✓ ResultMsg

Figure 5.1: The network graph G (left) and the new topology created by the XTC algorithm (right).

not suitable in the way it is currently implemented. G_{xtc} changed very often by adding new link and removing other ones. A collection of four different topology graphs is shown in Figure A.2. The reason for this is that the measured RSSI value varied strongly. This lead to frequent changes of the orders at the nodes which resulted in continuous changes in G_{xtc}.

5.2 Dynamic Source Routing Results

5.2.1 Test Setup

The nodes were placed at the same positions as described in Section 5.1.1. Likewise, the same parameters were used for the XTC algorithm: A power level of 128 and the average of the two RSSI values as link quality metric.

Two separate experiments were performed: In the first experiment, the DSR algorithm was run on top of the XTC algorithm, i.e. it was only allowed to use the links of G_{xtc}. In the second experiment, the DSR algorithm was directly run on the network topology without using XTC.

In both experiments, the DSR algorithm performes equivalently: A node randomly selects a destination[1] and starts the route discovery phase. If a route is found, a payload message is sent to the source. After the acknowledgement is received, an EEPROM entry is written. If no acknowledgement is received within a certain period of time, there are up to two retries to retransmit the payload package over the known route. When the result is negative, a new route to the destination is searched. If this process failes three times — a total of nine failed attempts to reach the destination — the destination is considered unreachable. Such an unavailable route search was saved in the EEPROM, too.

After one iteration of the DSR algorithm, the node idled a random time between 0.5 and 4 seconds. After that period, it selected a new random destination and restarted the route discovery phase.

The setup for both experiments was very similar: The nodes were placed and turned on. In the first few minutes, the nodes were running only the XTC algorithm (Experiment 1) or sleeping (Experiment 2). This enabled the XTC

[1]The total number of nodes — 12 in this experiment — was given at compile time.

algorithm to build a (more or less) stable topology. After that, the dynamic source routing algorithm was started with a flooding message from the gateway node (see Figure A.1). After five minutes, the DSR algorithm (including XTC in the first experiment) was stopped using another flooding message. Finally, the data logged in the EEPROM was collected.

5.2.2 Results

In both experiments, a small number of nodes did not produce useful results and sometimes not even data at all. In Experiment 1, the results of node 4, 9, and 11 could not be read due to problems with the SerialForwarder. In Experiment 2, node 4 and 12 did not find any route, i.e. there were about 40 entries reporting a failed route search. The reason for this remains unknown, especially because other nodes could successfully route to these nodes and therefore they were able to communicate with the rest of the network and were not disconnected.

The results of the remaining nodes were very similar within the same experiment. The following table shows the results of node 6. It was one of the nodes that provided useful results in both experiments and it is therefore suitable for a direct comparison.

	Experiment 1	Experiment 2
Total number of route discoveries:	45	49
Number of failed route discoveries:	11 (24%)	3 (6%)
Average route length:	2.18 hops	1.26 hops
Average RSSI per hop:	82.8	133.1
Maximum route length:	4 hops	3 hops

It is important to note that the amount of data collected is probably not big enough for significant conclusions. Additionally, due to the fact that the RSSI is not a reliable metric in the form it is used now (see Section 5.1.2), it will be difficult if not impossible to reproduce these measurements. However, since all nodes produced similar data, some interesting observations can be made:

- The number of failed route discoveries increases when using G_{xtc} as an underlying network topology. This is most probably due to the fact that G_{xtc} is not very stable for a long period of time. When one link is removed from G_{xtc}, all routes containing this link are broken in the first experiment. Since four messages on each link of a route are required for a successful data transmission, an instable topology graph significantly reduces the number of succeeded transfers. This effect could be reduced by using more stable link quality metrics.

- The average route length increases when routing on G_{xtc}. Since G_{xtc} is a subgraph of the complete network topology, this result is no surprise.

- The average RSSI per hop was improved by using XTC. Informally speaking, this means that better links were taken in the routes found in the first experiment. This is a strength of the approach taken in this experiment: Links with a high RSSI value (i.e. bad links) are removed from the topology. In doing so, the route length increases, but should be more stable.

Summing up the results, the two experiments confirmed our expectations that the average route length increases while the average RSSI per hop decreases. Unfortunately, the effect of using stable routes was nullified by the instability of G_{xtc}. The number of failed route discoveries should reduce by using only a subset of the network topology containing only “strong” links.

Chapter 6

Conclusion

6.1 The Program

The experiments have shown that the implementation of the XTC algorithm works correctly in a real environment with random packet loss. The application is able to dynamically adapt to changes of the underlying network topology. Additionally, the *XTC Interface* application displays the results of the algorithm on a PC.

6.2 Open Problems and Possible Extensions

During the work on this diploma thesis, many ideas for extension were proposed. While a lot of them were directly implemented, some of them had to be postponed to later projects due to time constraints. Additionally, the experiments have shown that some aspects were not perfectly solved in the current version of the implementation. These extensions and problems are presented in the following sections.

6.2.1 Link Quality Metrics

Using RSSI as link quality metric as it is done now is not sufficient as the experiments have shown — at least for indoor scenarios. Since the RSSI measurements strongly vary without changing the setup of the nodes, further action has to be taken. Possibilities include:

- Average the RSSI over multiple packets. For example, the RSSI of each incoming packet could be saved in a circular buffer. Instead of using `OrdCalcMsg` messages, an average of these values could be used.

- Create RSSI groups, each covering a range of the specturm of possible RSSI values. Two values are then considered equivalent if they are in the same group. Thereby, a variation of the measured RSSI values between the limits of its group would not affect the final link quality. This approach could be extended in a way that the RSSI group is only changed if multiple consequent measurements lie outside of the current group.

- Use of the RSSI value together with other parameters like the error rate or the delay of a link. That way, variations of the RSSI would be compensated by the other factors.
- Evaluate the link quality as it is done now, but improve the quality value for links which are currently part of G_{xtc}. In doing so, existing links would be preferred to other links with a similar quality value which are not part of G_{xtc}, reducing frequent changes in the result. Using the approach of artificially adjusting the link quality after the RSSI measurement, other features could be implemented. For example, a node with a high degree in G_{xtc} could lower the measured quality to prevent increasing its degree even more.

6.2.2 Packet Loss and High-Degree Networks

While the current implementation work flawlessly in networks where the degree of each node is limited to about ten, there are some issues when a node has a higher degree:

- When a node with high degree broadcasts an `OrdChangedMsg` message, it is possible — depending on the state of its neighbors — that it gets flooded with requests by its neighborhood. This leads to high packet loss and delays which probably result in temporary inconsistencies in the resulting topology graph.
- In the current implementation, `OrdChangedMsg` messages are only broadcasted, but not acknowledged. If such a message is lost or not received by a node, one or more nodes do not request the updated order of the node that changed. This may result in inconsistencies until this node sends another `OrdChangedMsg`. This problem could be solved by attaching a version number to each order. A node could then ask its neighbors for the current version number of their order on a regular basis.
- A node with a high degree has to partition its order into multiple parts due to the limited size of radio messages. In the current implementation, there are some problems with the retransmission and reconstruction of fragmented orders if certain packets are lost. In the worst case, these issues can completely block a node by switching to an illegal state with no possibility to recover with the exception of a manual reboot. These scenarios must be considered in more detail and the implementation needs to be adjusted to correctly handle them. The most elegant solution would be to create a separate network message layer which allows sending bigger packets by automatically splitting, transmitting, and reconstructing them.

6.2.3 Miscellaneous

Besides the points already mentioned, there are some smaller features that could be added:

- When the order of a node changes, this does not necessarily affect all its neighbors. For example, when the order of node u changes from $r < s < t$

to $r < t < s$, this change concerns only the common neighbors of the three nodes u, s, and t. All other neighbors of node u do not need to adjust their stored order of u. Because of that, appending a list of nodes which changed their position in the order of a node to each `OrdChangedMsg` message would reduce the number of messages.

- The current implementation of the Dynamic Source Routing does not cache the routes found for later reuse. In doing so, the performance of the algorithm could be increased. Besides that, there are countless possibilities to improve the DSR algorithm, some of which are mentioned in Section 4.1.

- The *XTC Interface* could be extended with many features. For example, it would be convenient to have the possibility to add a map like the one showed in Figure A.1 in Appendix A as background picture of the application. The nodes could then be easily placed according to their real position on the map. Another handy feature would be that the nodes are automatically placed by the application in a better way than it is done now. One approach to implement such an attribute could use a spring model based on measured link quality.

6.3 Personal Experience

During all stages of this diploma thesis — from the first contact with TinyOS and nesC over the design and implementation phase until the final tests and measurements — it was very exciting to work on this project. It was interesting to gain experience with small embedded platforms where the limited resources impose different approaches to solutions than on architectures like PCs which are encountered during normal studies. Additionally, I am fascinated by the concept of distributed units working together with very limited local knowledge and without having an idea about what is happening overall.

Of course, there were also some less enjoyable parts during this project. This mainly includes the debugging since it is very hard to understand what is happening inside a sensor node when the LEDs are flashing in unexpected combinations or the node does not do anything at all. Unfortunately, there are almost no tools available to help a developer reproducing the activity of a node. Besides these software issues, the hardware itself occasionally astonished with unexpected behavior. While this could be tracked down to weak batteries in some cases, the reason for most surprises remains unknown, especially because the motes later switched back to normal operation without any changes.

All together, I do not regret choosing this topic for my diploma thesis and I will keep it in memory as a very positive experience.

Appendix A

Performance Evaluation Results

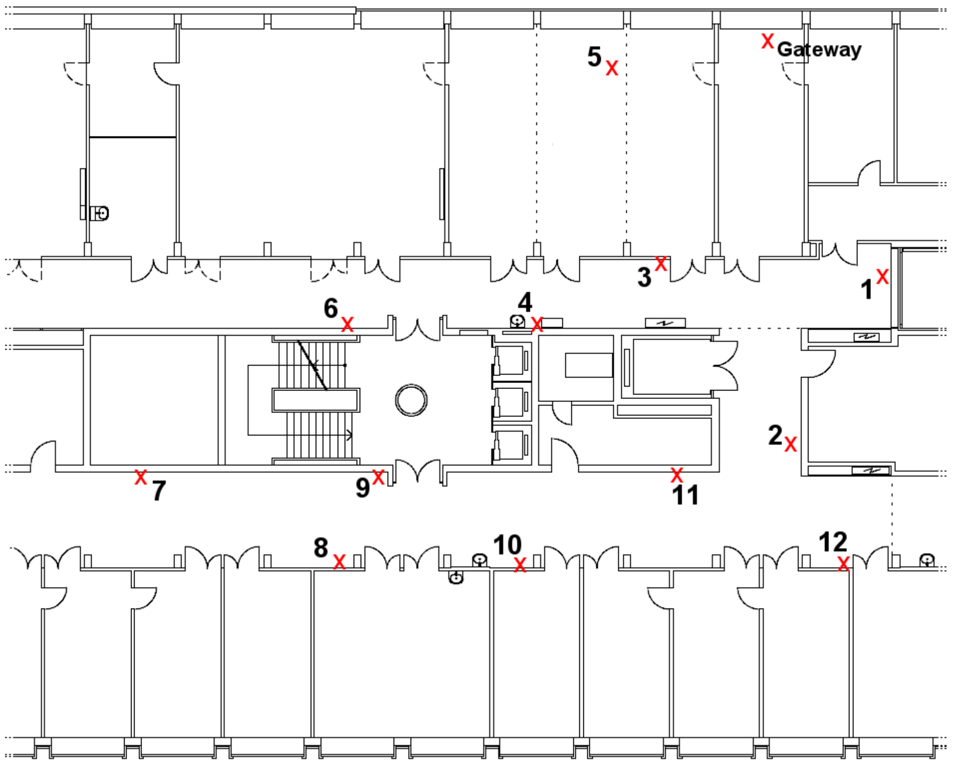
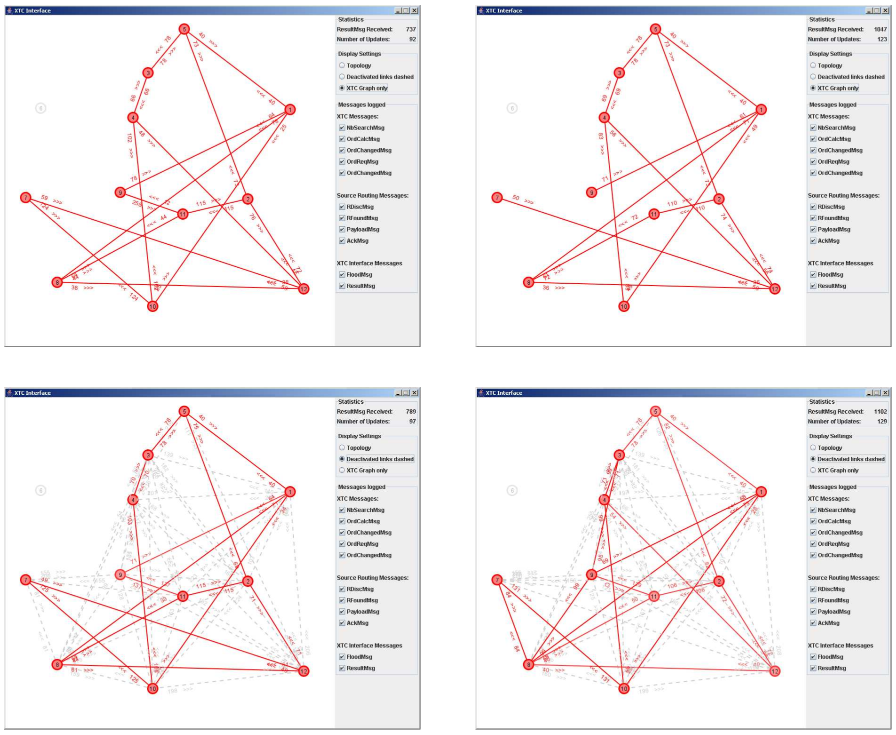


Figure A.1: The position of the twelve nodes for the experiments done in Chapter 5.


XTC Interface

Statistics
ResultMsg Received: 737
Number of Updates: 92

Display Settings
Topology
Deactivated links dashed
[✓ XTC Graph only]

Messages logged
XTC Messages:
✓ NbSearchMsg
✓ OrdCalcMsg
✓ OrdChangedMsg
✓ OrdReqMsg
✓ OrdChangedMsg

Source Routing Messages:
✓ RDiscMsg
✓ RFoundMsg
✓ PayloadMsg
✓ AckMsg

XTC Interface Messages
✓ FloodMsg
✓ ResultMsg

XTC Interface

Statistics
ResultMsg Received: 1047
Number of Updates: 123

Display Settings
Topology
Deactivated links dashed
[✓ XTC Graph only]

Messages logged
XTC Messages:
✓ NbSearchMsg
✓ OrdCalcMsg
✓ OrdChangedMsg
✓ OrdReqMsg
✓ OrdChangedMsg

Source Routing Messages:
✓ RDiscMsg
✓ RFoundMsg
✓ PayloadMsg
✓ AckMsg

XTC Interface Messages
✓ FloodMsg
✓ ResultMsg

XTC Interface

Statistics
ResultMsg Received: 789
Number of Updates: 97

Display Settings
Topology
[✓ Deactivated links dashed]
XTC Graph only

Messages logged
XTC Messages:
✓ NbSearchMsg
✓ OrdCalcMsg
✓ OrdChangedMsg
✓ OrdReqMsg
✓ OrdChangedMsg

Source Routing Messages:
✓ RDiscMsg
✓ RFoundMsg
✓ PayloadMsg
✓ AckMsg

XTC Interface Messages
✓ FloodMsg
✓ ResultMsg

XTC Interface

Statistics
ResultMsg Received: 1102
Number of Updates: 129

Display Settings
Topology
[✓ Deactivated links dashed]
XTC Graph only

Messages logged
XTC Messages:
✓ NbSearchMsg
✓ OrdCalcMsg
✓ OrdChangedMsg
✓ OrdReqMsg
✓ OrdChangedMsg

Source Routing Messages:
✓ RDiscMsg
✓ RFoundMsg
✓ PayloadMsg
✓ AckMsg

XTC Interface Messages
✓ FloodMsg
✓ ResultMsg

Figure A.2: Four different topology graphs produced during the experiment. The dashed grey lines in the two pictures at the bottom show the complete topology. The delay between two pictures varies between a half and two minutes.

Bibliography

[1] N. Burri, R. Wattenhofer, Y. Weber, and A. Zollinger. SANS: A Simple Ad hoc Network Simulator. In *Proc. of the World Conference on Educational Multimedia, Hypermedia & Telecommunications (ED-MEDIA)*, June/July 2005.

[2] B. N. Clark, C. J. Colbourn, and D. S. Johnson. Unit Disk Graphs. *Discrete Mathematics*, 86:165-177, 1990.

[3] Crossbow Technology Inc.
`http://www.xbow.com`

[4] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proc. the* 2^{nd} *Int. Conferences on Embedded Network Sensor Systems (SenSys)*, 2004.

[5] nesC 1.1 Language Reference Manual
`http://nescc.sourceforge.net/papers/nesc-ref.pdf`

[6] R. Prakash. Unidirectional Links Prove Costly in Wireless Ad-Hoc Networks. In *Proc. of the* 3^{rd} *Int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M)*, 1999.

[7] TinyOS Homepage
`http://www.tinyos.net`

[8] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu: The Broadcast Storm Problem in a Mobile Ad Hoc Network. *Wireless Networks*, 8:153-167, 2002.

[9] R. Wattenhofer and A. Zollinger. XTC: A Practical Topology Control Algorithm for Ad-Hoc Networks. In *Proc. of the* 4^{th} *Int. Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN)*, April 2004.