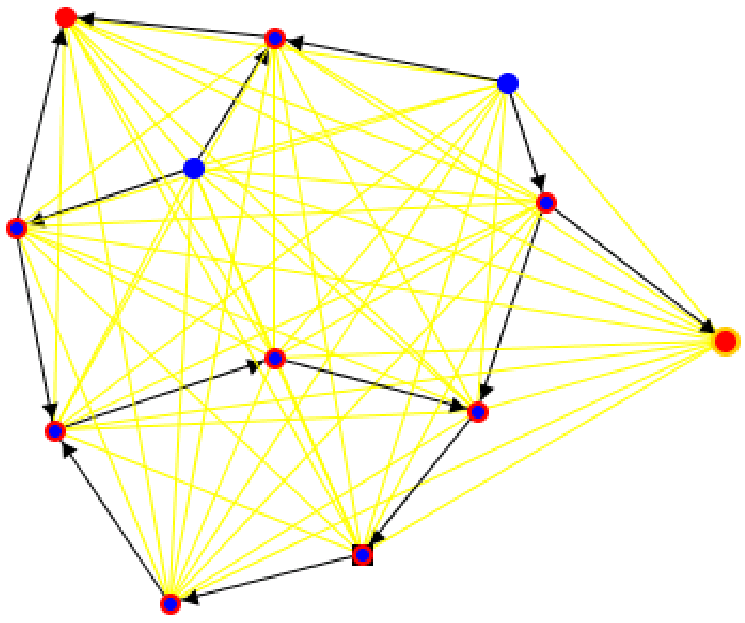
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Adaptive XTC on BTnodes

Kevin Martin



Masterarbeit
MA-2005.05

Wintersemester 2004/2005

Betreuer: Matthias Dyer
Jan Beutel
Professor: Lothar Thiele

Preface

Essentially, I've chosen this master thesis because of its variety: On one side there was the challenge of implementing a newly proposed topology control algorithm on a limited hardware platform, which requires simple but efficient compromises to come along with the restrictions given by the hardware platform.

On the other side there was the XTC topology control algorithm which was mainly developed for network establishment, thus being of wide scope for the development of own ideas concerning network maintenance. Hence, this thesis was not only about the implementation of an algorithm on the BTnode - for me, the theoretical aspects this thesis had to offer played a key role in my decision as well.

Last, but not least, it was also the intuitive behavior of the XTC algorithm: when drawing a sample constellation of unconnected network nodes, having in mind to form a sparse but robust, connected network graph, I always end up in a solution quite similar to the XTC solution which is really impressive.

I give my sincere thanks to my advisor, Matthias Dyer for his support and encouragement, as well as for his guidance and distributions on my thesis. I'm also very thankful to my co-advisor, Jan Beutel for his valuable suggestions and encouragement.
My gratitude goes to the people at the TIK laboratory, especially to Prof. Lothar Thiele who made this thesis possible.

Contents

List of Figures

Abstract

With the XTC algorithm, a new, yet extremly simple topology control algorithm for ad hoc networks has been proposed. In contrast to previously proposed algorithms XTC is strictly local, does not require availability of node position information, and proves correct also on general weighted graphs. Additionally, XTC is targeted to resource constrained nodes which makes it to one of the currently most realistic topology control algorithm available.

In this thesis, an adaptive variant of the XTC algorithm has been developed and implemented on the BTnode, an autonomous wireless communication platform based on a Bluetooth radio. Basically, the adaptiveness is achieved by letting the nodes periodically establish links to unconnected neighboring nodes that seem to be "better" than the set of the currently chosen links. In the same way "bad" links are identified and closed. Although inefficient in its rough draft, power consumption due to excessive node activity can be reduced drastically by observing unestablished links.

On the basis of this implementation, the XTC topology control graph hasy been successfully established within a larger deployment of 39 BTnodes in an indoor environment. Additionally, it is shown that the XTC algorithm implemented on a real-world platform is highly sensitive to fluctuating and asymmetric link weigths, and that the strict locality of the XTC algorithm gives the BTnodes a lot of trouble during network establishment in higher density networks. Furthermore, since the XTC algorithm does not regard scatternet formation explicitly, the correctness of the resulting topology control graph cannot be guaranteed without any further actions taken. Hence, to solve these problems a lot of engineering efforts had to be taken which is explained in this thesis.

Chapter 1

Introduction

1.1 Motivation

In the research field of ad-hoc networking, various different algorithms have been developed for reducing power consumption to guarantee maximal lifetime of the network nodes. Most of them are developed and evaluated in theory, come along with a lot of of promising properties, and can be formulated in a few lines of pseudo-code.

Anyhow, in practice an implementation of most of the algorithms proposed has never been presented because it turned out that even the apparently simplest algorithms would result in implementations way to complex for currently available platforms. In fact, the simplicity of algorithm formulations mostly is achieved by implicitly making assumptions that do not hold in practice.

With the XTC algorithm, a new algorithm for network establishment and topology control has been proposed that seems to be extremely simple. Furthermore, the algorithm promises to produce a mesh-like network topology which seems to be a good compromise between connectivity and sparseness. Thus, implementing a dynamically adapting variant of the XTC toplogy control algorithm seems to be a very realistic approach to obtain an implementation that covers all of our demands for topology control.

Basically, the target was to derive a dynamically adapting variant of the available XTC algorithm and implement it on an existing platform to obtain first results of both, the available XTC algorithm, as well as of the dynamically adapting variant derived in this thesis, concerning their performance, their limitations, as well as their behaviour and usability in a real-world system.

Anyhow, the real challange of this thesis was to map an apparantly simple topology control algorithm for ad-hoc networks onto an existing platform, discovering the limitations of the algorithm, of the hardware platform, and their interaction. As not expected, a lot of engineering effort had to be done to obtaine a well functioning, robust implementation of the XTC algorithm.

1.2 Ad-hoc networks

A mobile ad-hoc network (MANET) is an autonomous system of mobile nodes, (arbitrarily) connected by wireless communication links. In contrast to e.g. today's cellualar phone networks or wireless local area networks (WLAN's), mobile ad hoc networks do not make use of a backbone network consiting of fixed centralized nodes. Messages are exchanged using only the communication links between mobile nodes, i.e. messages between two nodes are exchanged either directly or by making use of several relay nodes.

Mobile ad-hoc networks may operate in a stand alone fashion, that is the mobile network nodes are able to establish and maintain a communication network dynamically and without the need of human interaction. Thus, in contrast to a fixed wireless network, a MANET can be deployed in any geographical location, requiring minimum setup and administration costs.

1.2.1 Topology Control Algorithms

The primary target of a *topology control algorithm* is to establish connections between mobile nodes in a way that the union of the links established form a connected network. Additionally, the topology of the resulting network shall be energy conserving, the lifetime of the network nodes by shall be increased by "wisely" connecting the network nodes.

The most popular requirements upon the resulting network topology are:

1. *Connectivity* The network should be robust to link losses, that is if a device leaves the network (e.g. due to power-down), the remaining nodes should still be able to communicate.

2. *Sparsness* The resulting network graph should be sparse, that is the number of links should be in the order of the number of nodes. Thus, in general there's a trade-off between sparseness and connectivity.

3. *Low Degree* Each network node has only a small number of communication links. In particular the maximum degree should be bounded from above by a constant.

4. *Avoidance of Long-Distance Links* Instead of creating power consuming long-distance communication links, messages should rather be routed over several small (energy-efficient) hops.

1.2.2 XTC Algorithm

The XTC algorithm [1] is an extremly simple, strictly local topology control algorithm for mobile ad-hoc networks. In contrast to previously proposed algorithms, the XTC topology control algorithm does not require availability of node position information. Instead, XTC operates with a general notion of order over the neighbor's link qualities.

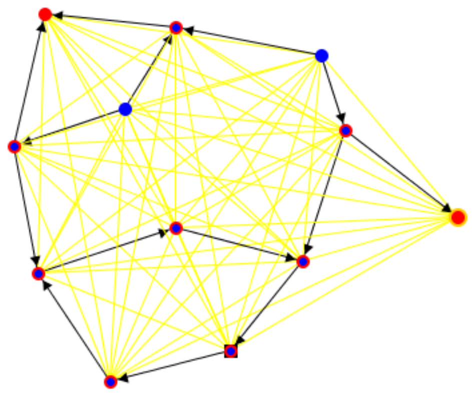


Figure 1.1: Mesh Character of XTC topology control graphs

Additionally, the resulting network topology is proven to be sparse, as well as to have bounded degree. In fact, XTC promises to establish a sparse network graph with mesh character as shown in figure 1.1, avoiding long-distance communication links. This seems to be an excellent compromise between connectivity and sparesness which makes XTC - together with its simplicity - extremly attractive.

1.2.3 Bluetooth Networks: Scatternets

The basic unit of networking in Bluetooth is a piconet [2], consisting of a master and from one to seven active slave devices, as shown in figure 1.2 a). The device designated as the master makes the determination of the channel (frequency-hopping sequence) and phase (timing offset that is, when to transmit) that will be used by all devices on this piconet.

A device in one piconet may also exist as part of another piconet and may function as either a slave or master in each piconet. This form of overlapping is called a *scatternet* (see figure 1.2 b).

Bluetooth scatternets come along with some restrictions:

1. A Bluetooth piconet is limited to consist of eight devices, i.e. a bluetooth master device may not be connected to more than seven slave devices.

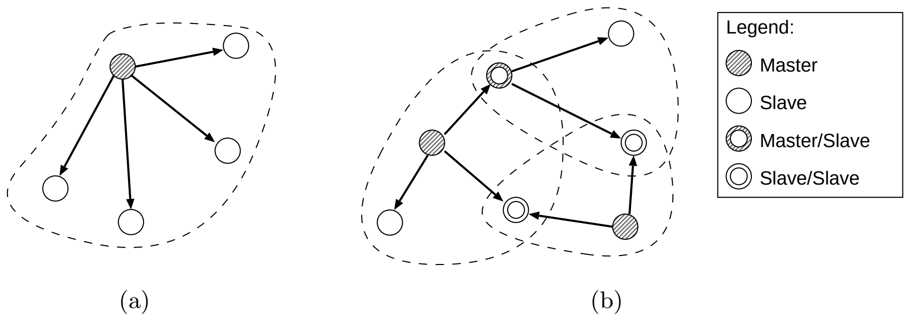


Figure 1.2: Bluetooth networks: Piconet (a), and multiple piconets connected to a scatternet (b).

2. A Bluetooth device may not be part of more than four piconets, thus a bluetooth device may not be a slave device of more than three different master devices.

Therefore, a Bluetooth device is not able to maintain more than ten connections at once, which means that network graphs with degree > 10 may not be established with Bluetooth devices.

1.3 The BTNode

The BTnode is an autonomous wireless communication and computing platform based on a Bluetooth radio and a microcontroller (figure 1.3). It serves as a demonstration platform for research in mobile and ad-hoc connected networks. The BTnode has been jointly developed at the ETH Zurich by the Computer Engineering and Networks Laboratory (TIK) and the Research Group for Distributed Systems.

1.3.1 System Overview

The BTnode rev3 is a dual radio device compatible to the old BTnode rev2 and the Berkeley Motes. An overview of the BTnode system is given in figure 1.4.

The BTnode rev3 features a Zeevo ZV4002 Bluetooth system supporting up to 4 independent Piconets and 7 Slaves and an additional Chipcon CC1000 low-power radio. The system is built around an Atmel ATmega128l microcontroller with an integrated battery case to house 2xAA cells and extension connectors. The BTnode is built on a 4 layer PCB measuring 58.15x32.5mm.

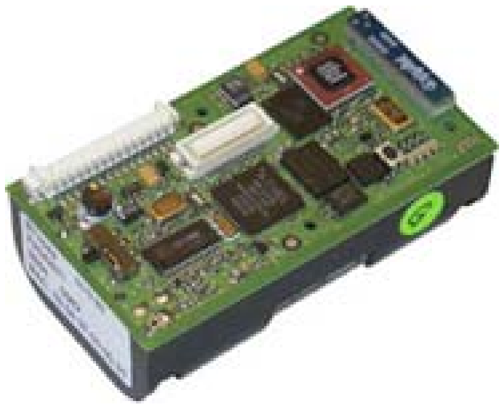


Figure 1.3: The BTnode

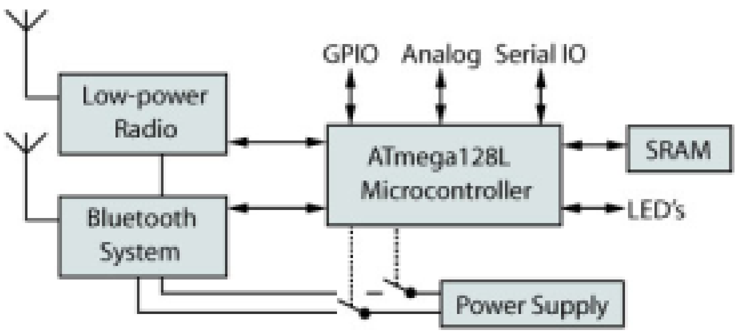


Figure 1.4: BTnode System Overview

1.3.2 Nut/OS

The BTnodes run an embedded systems operating system from the open source domain - the Nut/OS. Basically, Nut/OS is an intentionally simple real-time operating system for the Atmel ATmega128 microcontroller, which provides a minimum of network oriented system services. It's features include:

- Non preemptive cooperative multi-threading
- events
- Periodic and one-shot timers
- Dynamic heap memory allocation
- Interrupt driven streaming I/O

1.4 Connection Manager

The *connection manager* is a software component running on the BTnodes, which is responsible for the discovery of other nodes and the creation and maintenance of a Bluetooth Scatternet. In other words, a *connection manager* is just an implementation of a topology control algorithm.

Basically, the connection manager was intended to be part of the JAWS application [6], but due to its simple interfaces it can be used by different applications too.

As an example, the modular structure of the JAWS application is shown in figure 1.5: each time the connection manager established / closed a connection, the higher layer gets signaled by calling a callback function. This callback function can be registered at the connection manager by the application layer (in this case the transport layer).

1.5 Target

The target of this thesis was to develop a XTC based connection manager, i.e. the XTC topology control algorithm should be implemented on the BTnodes. On the basis of this implementation, the practical ability of the XTC algorithm should be estimated. More precisely, this includes

1. an evaluation of its implementation complexity
2. testment of its properties in a real-world deployment
3. evaluation of its practical limitations

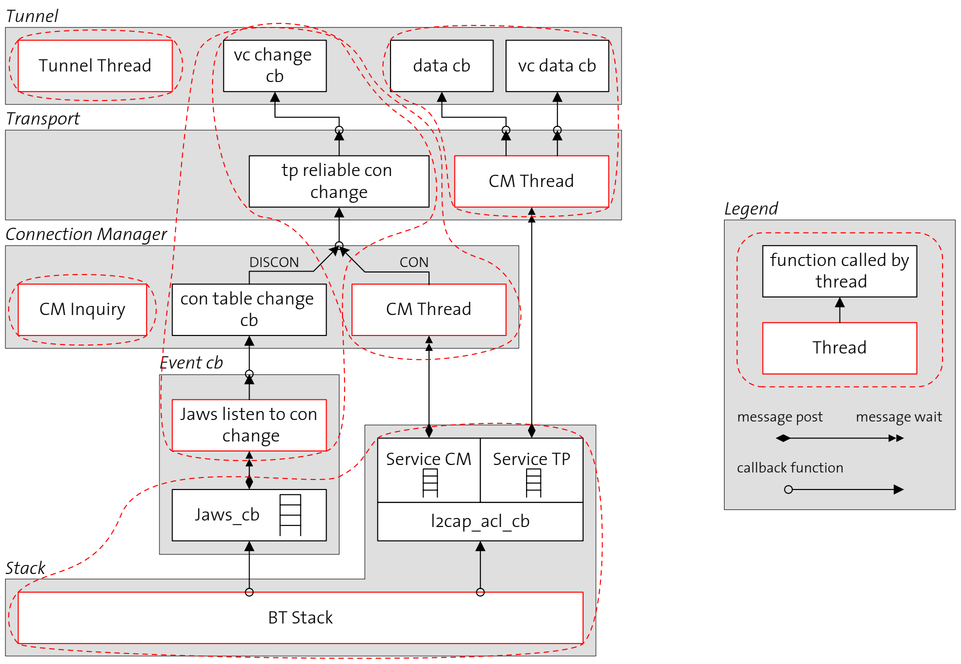


Figure 1.5: JAWS Components

Since the XTC algorithm as published in [1] considers network establishment only, another important part of this thesis was to find solutions for maintaining an XTC established network. More precisely, the XTC algorithm had to be extended for that the topology of the network is adaptive, that is the network should be be able to deal with

1. repositioning of network nodes
2. leaving devices (e.g. due to power-down)
3. entering devices

The original problem task is given in appendix C.

1.6 Related Work

In [7] a scalable topology control algorithm for deployment-support networks (DSNs) is presented which forms a tree-topology. Due to the topology formed by this algorithm, explicit route calculations are not necessary since there's only one path between any two nodes. Anyhow, the tree topology is not robust to link losses, and the number of relay nodes needed to send a message from some node to a remote node may be excessively large since cycles are avoided. Furthermore, the topology algorithm proposed does not abandon long-distance links.

In [9] several scatternet formation protocols for Bluetooth devices have been proposed and evaluated. In fact, these evaluations are based upon simulation results, but until today, none of the algorithms proposed has been implemented in a real-world system.

1.7 Overview

This thesis is organized as follows. In chapter 2 an introduction to the XTC algorithm as published by Wattenhofer at al. in [1] is given, including some alternative interpretations of the XTC link selection procedure. During the second chapter an iterative variant of the XTC algorithm is derived, which is necessary for that the XTC algorithm can be mapped onto a hardware platform that has to make use of neighbor discovery and connection establishment, as it is the case for the BTnode.

In the third chapter, a variant of the XTC algorithm is proposed that is able to adapt the topology of the network dynamically if significant changes occur (e.g. repositioning of some node). Furthermore, the problems arising with asynchronous link weight updating, as well as the problem of link losses are discussed.

Afterwards, an overview of the implementation is given (chapter 5. Implementation details are given in the following chapter, which additionally points out that implementation complexity of the adaptive XTC topology control algorithm is not that low.

In chapter 7 we present the results obtained from with implementation, while chapter 8 concludes this thesis.

Chapter 2

The XTC Topology Control Algorithm

This chapter is all about the XTC Topology Control algorithm. In the first section we give formal definitions of basic concepts used throughout this thesis. Afterwards, the XTC topology control algorithm as published by Wattenhofer et al. [1] gets introduced, followed by a summary of its most important properties. To finish this introduction of the XTC algorithm, an example is given in section 2.4.

In the last section of this chapter, we're going to give some alternative interpretations of the XTC link selection procedure. More precisely, we're going to show that in XTC

1. link selection can be done without storing any information received by neighbor nodes,
2. a link is discarded only if a better common neighbor exists.

These interpretations are the basis for deriving the iterative XTC algorithm, as well as the adaptive XTC algorithm presented in the following two chapters.

2.1 Preliminaries

The ad-hoc network before running the topology control algorithm is denoted as $G = (E, V)$, with V being the set of ad-hoc network nodes, and E representing the set of communication links. There is a link (u, v) in E if and only if the two nodes u and v can communicate directly. Running the XTC topology control algorithm will yield a sparse subgraph $G_{XTC} = (V, E_{XTC})$ of G, where E_{XTC} is the set of remaining links.

In a *weighted graph* $G = (V, E)$ every edge $(u, v) \in E$ is attributed a weight ω_{uv}. When referring to a weighted graph, we assume that the weights are symmetric: $\omega_{uv} = \omega_{vu}$. A completely unconnected weighted graph $G = (V, E)$ with $E = \emptyset$ is denoted by G_0.

The nodes of a *Euclidean graph* are assumed to be located in a Euclidean plane. Furthermore, the edge weight of an edge (u, v) is defined to be $\omega_{uv} = |uv|$, where $|uv|$ is the Euclidean distance between the nodes u and v. Note that the definition of Euclidean graphs does not contain a statement on the existence of certain edges.

A *Unit Disk Graph* is a Euclidean graph containing an edge (u, v) if and only if $|uv| \leq 1$.

2.2 Description

This section describes the XTC topology control algorithm as it was published by Wattenhofer et al. in [1].

Basically, the XTC algorithm consists of three main steps:

I Neighbor ordering

II Neighbor order exchange, and

III Link selection.

In the first step each network node u computes a total order $\prec_u$ over all its neighbors in a given network graph G. This order is intended to reflect the quality of the links to the neighbors.

A node u will consider its neihgbors in G (in the third step of the algorithm) according to $\prec_u$ ordered with respect to decreasing link quality: The link to a neighbor appearing early in the order $\prec_u$ is regarded as being of higher quality than the link to a neighbor placed later in $\prec_u$[1]. A neighbor w appearing before v in $\prec_u$ is denoted as $w \prec_u v$.

For illustration purposes in this thesis we assume that $\prec_u$ corresponds to the order of the neighbor's Euclidean distances from u.

[1]If two links are of equal quality, the link to the neighbor with smaller identity is considered as the higher quality link

```
XTC Algorithm

  I Establish order $\prec_u$ over $u$'s neighbors in $G$

 II Broadcast $\prec_u$ to each neighbor in $G$; receive or-
    ders from all neighbors

III Select topology control neighbors:
    1: $N_u := \{\}$; $\widetilde{N}_u := \{\}$
    2: while ($\prec_u$ contains unprocessed neighbors) {
    3:     $v :=$ least unprocessed neighbor in $\prec_u$
    4:     if ($\exists w \in N_u \cup \widetilde{N}_u : w \prec_v u$)
    5:         $\widetilde{N}_u := \widetilde{N}_u \cup \{v\}$
    6:     else
    7:         $N_u := N_u \cup \{v\}$
    8: }
```

Figure 2.1: The XTC algorithm

In the second step the neighbor order information is exchanged among all neighbors: a node u broadcasts its own neighbor order while receiving the orders established by all of its neighbors.

During the third step, which does not require any further communication, each node locally selects those neighboring nodes which will form its neighborhood in the resulting topology control graph, based on the previously exchanged neighbor order information. For this purpose, a node u traverses $\prec_u$ with decreasing link quality: "Good" neighbors are considered first, "worse" ones later. Informally speaking, a node u only builds a direct communication link to a neighbor v if u has no "better" neighbor w that can be reached more easily form v than u itself.

Although the XTC algorithm is executed at all nodes, the detailed description as shown in figure 2.1 assumes the point of view of a node v. The lines with a leading small number from 1-8 define Step III) in more detail: First, the two sets $\widetilde{N}_u$ and N_u are initialized to be empty. Now the neighbor ordering $\prec_u$ established in Step I), is traversed in increasing order (that is in decreasing link quality). In Line 4 the neighbor order $\prec_v$ of the currently considered neighbor v is examined: If any of u's neighbors w already processed appears in v's order before u ($w \prec_v u$) node v is included in $\widetilde{N}_u$ (Line 5); otherwise v is added to N_u (Line 7).

After completion of the algorithm, the set N_u contains u's neighbors in the topology control graph G_{XTC}.

2.3 Properties

This section shall give a brief overview of the most important properties of a resulting topology control graph G_{XTC}. The theorems presented here are adopted from [1]. Proofs and further details are not presented here - they have to be gathered from [1].

Theorem 2.3.1 (Connectivity) *Given a general weighted graph G, two node u and v are connected in G_{XTC} if and only if they are connected in G. Consequently, the graph G_{XTC} is connected if and only if G is connected.*

This theorem simply states that a topology control graph G_{XTC} will be connected, i.e. it states that the resulting network will not be partitioned.

Theorem 2.3.2 *Given a general weighted graph G, G_{XTC} has girth 4, that is, the shortest cycle in G_{XTC} is of length 4.*

This theorem is important because it ensures that the resulting topology of G_{XTC} is a good compromise between connectivity and sparseness: an XTC topology control graph will contain cycles (thus improves robustness of the network to link losses), but the cycle length is lower bounded.

Theorem 2.3.3 (Bounded Degree) *Given a Unit Disk Graph G, G_{XTC} has degree at most 6.*

The bounded degree property makes the XTC algorithm especially attractive for topology controlling an ad-hoc networks consisting of BTnodes since bluetooth devices are not able to connect to more than 10 neighbor devices. Unfortunatly, it only holds for Unit Disk Graphs but not for general weighted graphs.

2.4 An Example

The most easy way to understand the XTC Algorithm is by applying it on a simple topology graph G as depicted in figure 2.2 a). To keep the following example as simple as possible, we choose the euclidean distance between two nodes as the quality measure of a link (the shorter the euclidean distance, the better is the link quality).

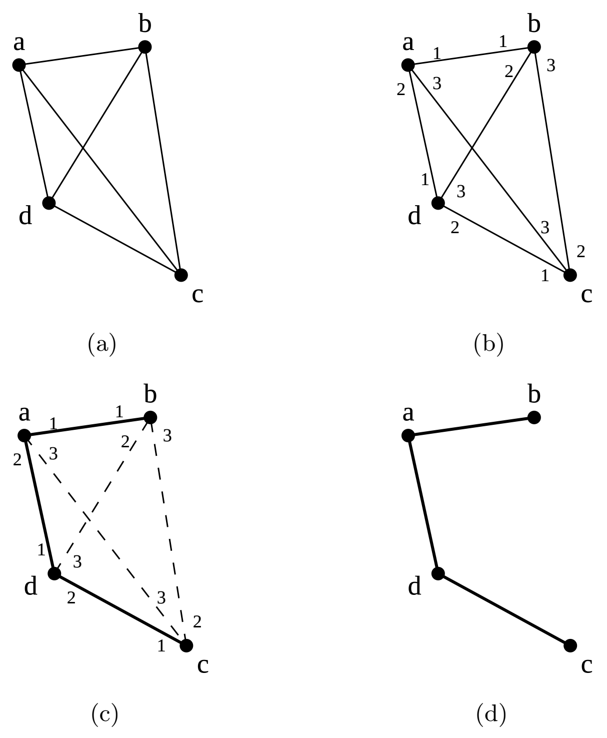


Figure 2.2: XTC: simple example

Applying the XTC Algorithm on the topology graph G shown in figure 2.2 a) - step by step - leaves to the following results:

1. First, each node establishes an order over its neighbor nodes as depicted in figure 2.2 b). As mentioned above, criteria of order establishment is the euclidean distance between two nodes. So if we take a look at e.g. node d, we see that node a is closest to node d, followed by node c which is closer to d than node b. This leads to the following order:

   $\prec_d$**:** a c b

   The orders established by nodes a, b, and c will look like this:

   $\prec_a$**:** b d c

   $\prec_b$**:** a d c

   $\prec_c$**:** d b a

2. During the second step of the XTC Algorithm, each node sends the order established to all its neighbor nodes. At the same time, each node receives orders from its neighbor nodes and collects them. Thus, the information collected e.g. by node a, choosing a tabular representation, will look like this:

$\boldsymbol{\prec_a}$**:**	b	d	c
	a	a	d
	d	c	b
	c	b	a

Table 2.1: Information collected by node A after XTC step 2

The representation shown in table 2.1 has to be interpreted as follows: the first row of the table represents the order established by node a (indicated by the box around a), and each column represents the order of the node in the corresponding header (e.g. the second row is the order established by node d).

3. In the third step, the nodes locally select the desired connections among all the possible connections as described in the link selection procedure (Step III in figure 2.1). From the point of view of a node u, this is done by examining the orders received, starting with the order of its "best" neighbor node (the first neighbor appearing in $\prec_u$). So in our example, from the point of view of node a, we first have to examine the order of node b, which is the first column of table 2.1:

$\boldsymbol{\prec_a}$**:**	b	d	c
	a	a	d
	d	c	b
	c	b	a

Reaching line 4 of the link selection procedure shown in figure 2.1 (Step III), we see that since the two sets N_u and $\widetilde{N}_u$ are empty, there cannot be a node w that appears before node a in the order $\prec_b$ of node b. Therefore, we add node a to the set N_u:

$$N_a = \{b\}, \qquad \widetilde{N}_a = \{\}.$$

Frome the point of view of node a, the next "best" neighbor in $\prec_a$, is node d: The set $N_a \cup \widetilde{N}_a$ now contains node b, hence we have to check if node b appears befor a in the order $\prec_d$ of node d. Using the tabular representation introduced earlier, that is, starting from top of the second column: if a appears before b we add d to N_a, which obviously is the case. This is shown in the table below:

$\prec_a$:	b	d	c
	a	**a**	d
	d	c	b
	c	b	a

Thus, we now have:

$$\widetilde{N}_a = \{\}, \qquad N_a = \{b, d\}.$$

The next "best" neighbor appearing in the order $\prec_a$ of a is also the last neighbor of a's neighbor ordering: node c. The set $N_a \cup \widetilde{N}_a$ now contains the nodes b and d, so we now have to check if b or d appears before a in the order $\prec_c$ of node c, that is, starting from top of the third column, we have to check if a appears before b and d:

$\prec_a$:	b	d	c
	a	a	d
	d	c	b
	c	b	**a**

As we can see, b, as well as d appear before a. Hence, we add c to the set $\widetilde{N}_a$ which gives us:

$$\widetilde{N}_a = \{c\}, \qquad N_a = \{b, d\}.$$

The set N_a now contains a's neighbors in the topology control graph G_{XTC}, that is the links (a, b) and (a, d) are part of the topology control graph G_{XTC}:

$$\{(a, b); (a, d)\} \in G_{XTC}.$$

Executing the XTC link selection procedure on each node depicted in figure 2.2 a) will lead to the topology control graph shown in figure 2.2 d).

2.5 Alternative Interpretations

In this section, we're going to present alternative interpretations of the XTC link selection procedure. We're going to see that neither is the sequence of processing the received orders relevant, nor is it necessary to store them. To derive this alternative interpretations we're going to take a closer look at the link selection procedure of the XTC algorithm presented in section 2.2 (figure 2.1).

2.5.1 Sequence independent Link Selection

We start with a somewhat more complex example as the example presented in the previous chapter: The constellation G_0 of unconnected network nodes before running the XTC algorithm on it is shown if figure 2.3 a).

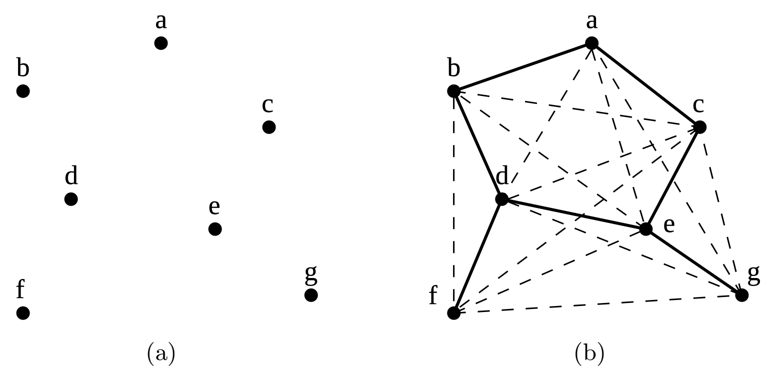


Figure 2.3: XTC: another example

The information received by node d after XTC Step II is shown in table 2.2 below, and we are interested in the decision made by the XTC link selection procedure about the link (d, e).

If we follow the XTC link selection procedure described in figure 2.1, we have to process the orders received from neighbors f and b (that is column one and two), before we can start processing order $\prec_e$ established by node e (third column). Therefore, when we start examining the order $\prec_e$ established by node e, the set $N_d \cup \widetilde{N}_d$ will contain the nodes f and b. Since neither f, nor b appears before d in the order of e (third row in table 2.2), we add e to the set N_d, which means that the link (d, e) will be part of the resulting topology control graph G_{XTC}.

$\boxed{\prec_d}$	f	b	e	a	c	g
	d	d	g	c	e	e
	e	a	c	b	a	c
	b	f	**d**	d	g	d
	g	e	a	e	d	f
	c	c	f	g	b	a
	a	g	b	f	f	b

Table 2.2: Information collected by node d after XTC step II, corresponding to the node constellation depicted in figure 2.3

It's easy to see that if we are going to process an order received by some node v, the set $N_d \cup \widetilde{N}_d$ always contains the nodes that are "better" than node v in $\prec_d$. Hence, by defining the set of superior nodes with respect to v:

$$S_{v,\prec_u} := \text{all neighbors that appear before } v \text{ in the order } \prec_u,$$

we can reformulate the link selection procedure as follows:

```
Select topology control neighbors:
1: while ($\prec_u$ contains unprocessed neighbors) {
2:     $v :=$ some unprocessed neighbor in $\prec_u$
3:     if ($\exists w \in S_{v,\prec_u} : w \prec_v u$)
4:         $(u, v) \notin G_{XTC}$
5:     else
6:         $(u, v) \in G_{XTC}$
7: }
```

From the equivalent formulation of the XTC link selection procedure shown above, it becomes clear that the sequence of processing the received neighbor orders is not relevant. Furthermore, from the point of view of some node u, deciding if a link (u, v) is part of G_{XTC} can be done independent of the orders received from the other neighbors.

2.5.2 Memoryless XTC

Since the sequence of processing the received neighbor orders is not relevant and can be done independently, it is not necessary to store the received orders. Instead, some node u can decide immediately after receiving a neighbor order $\prec_v$ if the link (u, v) should be part of the XTC topology control graph G_{XTC} or not. This allows us to give a somewhat more "reactive" interpretation of the XTC algorithm, shown in figure 2.4.

The advantages of the algorithm presented in figure 2.4 for implementation purposes are twofold:

```
XTC Algorithm

  I Establish order $\prec_u$ over $u$'s neighbors in $G$

 II Broadcast $\prec_u$ to each neighbor in $G$

III Select topology control neighbors:
    1: for (each received order $\prec_v$) {
    2:     if ($\exists w \in S_{v,\prec_u} : w \prec_v u$)
    3:         $(u, v) \notin G_{XTC}$
    4:     else
    5:         $(u, v) \in G_{XTC}$
    6: }
```

Figure 2.4: Alternative interpretation of the XTC algorithm

1. By implementing the algorithm shown in figure 2.1 straight forward, a node u would have to collect the neighbor orders of all neighbors and would then start processing them in proper sequence. Hence, we would have to make use of an inconvenient start condition like "neighbor orders received from all neighbors", which in practice may result in a unwanted approximation, since a node does not know when *all* orders have been received[2].

   By using the algorithm presented in figure 2.4, receiving a neighbor order can be viewed as an event that may occur at any time upon which a node has to react - either by adding the link to G_{XTC} or not (that is, keeping the link or closing it).

2. Some node u in G_{XTC} can repeat the link selection process, i.e. if an existing link (u, v) actually has to be in G_{XTC} or not can be checked by any node u at any time by sending the neighbor order to v. In section 4.6.2 we're going to see that this property is one of the key properties needed to react upon changes in topology.

3. Implementing the algorithm shown in figure 2.4 results in a more memory efficient implementation than an implementation of the algorithm presented in figure 2.1.

Because of its importance for the remainder of this thesis we reformulate the second reason mentioned above again:

[2] A similar problem is discussed in section 3.2 where a node has to discover "all" neighbor devices

Observation 2.5.1 *A link (u, v) that has to be removed from G_{XTC} due to a change in the topology graph G gets closed if either u or v resends its updated order.*

2.5.3 Better Common Neighbors

We are now going to give another interpretation of the link selection procedure of the algorithm shown in figure 2.4 again. More precisely, we only rewrite the selection condition in line 2 of the selection procedure, as shown in figure 2.5.

```
XTC Algorithm

  I Establish order $\prec_u$ over $u$'s neighbors in $G$

 II Broadcast $\prec_u$ to each neighbor in $G$

III Select topology control neighbors:
    1: for (each received order $\prec_v$) {
    2:     if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$)
    3:         $(u, v) \notin G_{XTC}$
    4:     else
    5:         $(u, v) \in G_{XTC}$
    6: }
```

Figure 2.5: "Better common neighbor" interpretation of the XTC algorithm

Basically, this is the same formulation as before, but the link selection condition of the algorithm shown in figure 2.5 is somewhat more intuitive than the condition of the algorithm shown in 2.1.

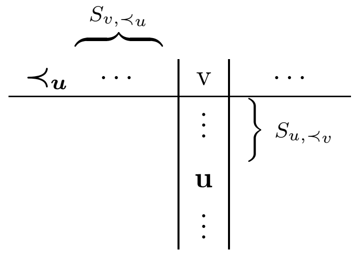


Table 2.3: Better common neighbor interpretation of the XTC link selection condition

This is shown in table 2.3: the set $S_{v,\prec_u}$ contains all the nodes that appear before v in the order established by node u, that is all the nodes appearing before node v in the header of the table. The set $S_{v,\prec_u}$ contains all nodes that appear before node u in the order established by node v, that is all nodes that appear before node v in the column with header v (starting from the top). Thus, the link selection condition can be interpreted as follows:

$$(u,v) \notin G_{XTC}, \text{ if a “better common neighbor” exists,}$$

or equivalently:

$$(u,v) \in G_{XTC}, \text{ if “no better common neighbor” exists.}$$

To give an example here, we refer back to the node constellation shown in figure 2.3 b): The link (d,a) is not part of the topology control graph G_{XTC} because there's the “better common neighbor” b. The table below shows how the “better common neighbor” can be identified in the table representation of the link selection condition:

$$\begin{array}{r|c|c}
\boldsymbol{\prec_d}\textbf{:}\ \ \overbrace{\mathrm{f}\ \ \underline{\mathrm{b}}\ \ \mathrm{e}}^{S_{a,\prec_d}} & \mathrm{a} & \cdots \\
\hline
 & \mathrm{c} & \left.\right\} S_{d,\prec_a} \\
 & \underline{\mathrm{b}} & \\
 & \mathrm{d} & \\
 & \vdots &
\end{array}$$

By underlining b, we've indicated that node b is the “better common node” of the link (d,a) and thus is the reason for that the link (d,a) is not in G_{XTC}.

Note that multiple “better common neighbors” may be the reason for that a link is not in G_{XTC}, e.g. if we take a look at the link (d,c) in figure 2.3 b) the link selection condition using tabular representation looks like this:

$$\begin{array}{r|c|c}
\boldsymbol{\prec_d}\textbf{:}\ \ \overbrace{\mathrm{f}\ \ \underline{\mathrm{b}}\ \ \mathrm{e}\ \ \ \underline{\mathrm{a}}}^{S_{c,\prec_d}} & \mathrm{c} & \cdots \\
\hline
 & \underline{\mathrm{e}} & \left.\right\} S_{d,\prec_c} \\
 & \underline{\mathrm{a}} & \\
 & \mathrm{g} & \\
 & \mathrm{d} & \\
 & \vdots &
\end{array}$$

We conclude the results obtained from the considerations done here with the following

Observation 2.5.2 *An edge (u,v) is not in G_{XTC} if and only if a better common neighbor $b_{u,v}$ exists.*

Chapter 3

Iterative XTC

The XTC algorithm introduced in the last section cannot be mapped directly onto our hardware platform - essentially because of the following two reasons:

1. Before two Bluetooth devices are able to communicate, a connection has to be established, and
2. before some Bluetooth device u is able to establish a connection to some Bluetooth device v, the desired communication partner v has to be discovered by device u.

Thus, before the XTC algorithm as published in [1] can be applied, we previously would have to established the completely connected network graph G, which is not desirable[1] Since this problem concerns not only Bluetooth devices but also every hardware platform that has to do neighbor discovery and connection establishement, we've decided to approach this problem on the algorithmic level. Thus, we're going to present a variant of the XTC algorithm in this chapter, which establishes the topology control graph G_{XTC} in an iterative fashion - starting with the completely unconnected network graph G_0

In section 3.1 we integrate connection establishments into the XTC algorithm shown in figure 2.5, section 2.5.3, while neighbor discovery is discussed in section 3.2. The iterative XTC algorithm will be introduced in section 3.3, and section 3.4 concludes this chapter.

[1]in fact, with Bluetooth devices it is not possible to establish a completely connected network graph $G = (V, E)$ with $V > 7$.

3.1 Connection Establishment

The XTC topology control algorithm as introduced in section 2.2 (figures 2.1, 2.4, and 2.5) starts with the completely connected network graph G. But in practice we have to start with the completely *unconnected* network graph G_0. Usually, we do not want to establish the completely connected topology control graph G before applying the XTC topology control algorithm - it is desired to close "unwanted" links as soon as possible[2]. Therefore, we have to include connection establishments explicitly into our formulation of the XTC algorithm.

3.1.1 Simple Protocol

The straight-forward way of including connection establishments in our XTC formulation of the XTC algorithm (figure 2.5) is presented in figure 3.1. Using this formulation we obtain the most simple XTC protocol possible, which is shown in figure (3.2).

```
XTC Algorithm

  I Establish order $\prec_u$ over $u$'s neighbors in $G_0$

 II Broadcast $\prec_u$ to each neighbor in $G_0$:
    1: for (each neighbor node $v$ in $G_0$) {
    2:     if ($\prec_v$ not yet received) {
    3:         establish connection to node $v$
    4:         send $\prec_u$ to node $v$
    5:     }
    6: }

III Select topology control neighbors:
    1: for (each received order $\prec_v$) {
    2:     if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$) disconnect
    3: }
```

Figure 3.1: XTC: alternative formulation, including connection establishment

A node u connects to one of its neighbor nodes u, and sends its order $\prec_u$ established during XTC Step II) to it. The receiving node v then immediately processes the order received, and decides if the link established should be kept or not (Step III). The link will be kept if the receiving node cannot find a better common neighbor b_{uv} in the order received and its own order.

[2]because in practice a node may only have a fixed number of open connections at once (e.g. the number of open connections with a BTnode is limited to 10)

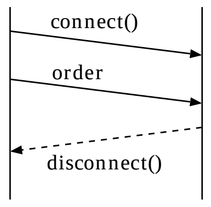


Figure 3.2: Simple XTC Protocol: time-line

If a better common neighbor can be found, the receiving node v will close the link (u, v).

The condition in Step II), Line 2 ensures that a link (u, v) won't be established twice if the link (u, v) is not in G_{XTC}: Assume that e.g. node u established a connection to node v, which decided to close the connection after receiving the order $\prec_u$ from node u. Since both, node u and node v are executing Step II) of the XTC algorithm, node v would establish the connection to node u again some moments later. But this is not necessary because of the symmetry of the XTC algorithm: Node u is going to make the same decision as node v, that is the connection will be closed again. Thus, by introducing the condition in Line 2, we can save some unnecessary connection establishments.

3.1.2 2-Way Handshake

It's obvious that with regard to simplicity and efficiency we can't do any better than with the simple protocol presented before. Nevertheless, this simple protocol comes along with some drawbacks:

Data Exchange The protocol depicted in figure 3.2 is unidirectional, i.e. the active side is able to send information to the passive side (that is sending the neighbor ordering plus some additional information if desired), but the passive side may not in case of a disconnection. In other words, if the receiving node decides to close the connection, there's no opportunity for it to give the initiator any further information (e.g. why the connection was closed).

Information Hiding An application layer should only be signaled about newly established connections if they are "wanted" (that is if they should be in G_{XTC}), thus connection establishments that are closed by the passive side right after receiving the neighbor ordering should be hidden from the application layer. To reach this, the passive side at least has to send back an acknowledgment for that the active side is able to signal the application layer.

Eliminating these drawbacks can be reached using the 2-Way Handshake protocol depicted in figure 3.3: a node u connects to one of its neighbor nodes and sends its neighbor ordering to it as before. Instead of just closing the link established or not, the receiving node sends back the result of the link selection decision, i.e. if the link should be kept, the receiving node sends back an acknowledgment to the sender ('ACK' packet). Also, if the receiving node decides that the link should be closed, the receiving node informs the sender about its decision (by sending back a 'NACK' packet), thus the initiator of the connection request is going to receive a response in any case.

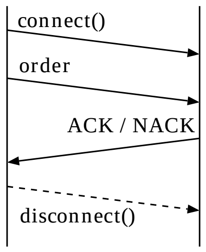


Figure 3.3: XTC protocol: 2-Way Handshake

By making use of the 2-Way handshake protocol presented in figure 3.3 the problems mentioned before are solved: Since the receiving node sends back a 'NACK' packet to the initiator of the connection request, it is able to add some additional information to the 'NACK' packet, e.g. the reason for its desire to close the connection[3].

Furthermore, hiding the connection request from the application layer can now be done by both parties: The initiating node informs the application layer not before receiving the 'ACK' packet, and the passive node right after an 'ACK' packet has been sent successfully.

[3]In section 4.6.2 we're going to see that giving one of the better common nodes as a reason for not acknowledging a connection request can be very useful for maintaining an XTC network

3.2 Device Discovery and XTC

The XTC topology control algorithm depicted in figure 2.1, as well as the algorithm presented in the previous section (figure 3.1) implicitly make the assumption that a given node u knows its neighborhood. That is, each node u knows which neighbor nodes are present, as well as how to build a connection to each of its neighbors. In Bluetooth this assumption is not realistic - a node u has to discover its neighborhood before the XTC algorithm can be applied.

The straight forward approach of concerning neighbor discovery in our XTC formulation would be to extend Step I) in the following way:

```
I Establish order $\prec_u$ over $u$'s neighbors in $G_0$:
  1: discover all neighbor devices of node $u$
  2: establish order $\prec_u$ over neighbor devices found
```

The problem of a formulation like this is that a function called *discover_all_neighbor_devices()* does not exist in practice: as we shall see, neighbor discovery is a very costly and time consuming process, and it is not possible to reliably discover "all" neighbors in a single discovery process - a node has to discover its surrounding step by step. More precisely: doing multiple consecutive neighbor discovery steps increases the probability of discovering all neighbor devices that are actually present. We are going to discuss this topic in more detail in section 6.1, chapter 6. The reason why we point out this behavior here becomes clear if we take a look at a more realistic formulation of XTC Step I:

```
I Establish order $\prec_u$ over $u$'s neighbors in $G_0$:
  1: **until** (all neighbors discovered) **do** {
  2:     discover new devices := $D_u$
  3:     $U_u$ := newly discovered devices
  4:     update neighbor ordering: $\prec_u = \prec_u \cup U_u$
  5: }
```

Line 2 indicates that the device has to start a new neighbor discovery process. The neighbor discovery result D_u may contain some devices that were not discovered so far; we denoted this set by U_u:

$$U_u := \{v \in D_u : v \notin \prec_u\}.$$

These newly discovered neighbor devices have to be added to the neighbor ordering $\prec_u$, which is done in Line 4. The first line indicates that each node has to do several neighbor discovery steps in order that "all" neighbor devices get discovered.

The problem mentioned above consists of the termination condition of the while loop (Line 1): in practice, it is not possible to implement the condition “all neighbors discovered”, since a node u does not know the number of neighbor nodes its neighborhood consists of. Therefore, we have to replace this condition, e.g. by the condition “N neighbors discovered” or the condition “N neighbor discovery steps performed”.

No matter how the termination condition gets chosen, the result is an approximation of the XTC algorithm that may cause unpredictable errors, as can be seen if we rewrite the whole XTC algorithm. This is shown in figure 3.4: The termination condition of Step I) has been replaced by just “some condition” which indicates that an approximation has to be made at that point.

In contrast to the algorithm presented in figure 3.1, we slightly modified Step II), which shall indicate that we can not guarantee that the order established by a node u can be sent to each neighbor in G_0, but instead is sent to all neighbor nodes that were found during Step I (i.e. to all nodes in $\prec_u$).

```
XTC Algorithm

  I Establish order $\prec_u$ over $u$'s neighbors in $G_0$:
    1: until (some condition) do {
    2:     discover new devices := $D_u$
    3:     $U_u := \{w \in D_u : w \notin \prec_u\}$
    4:     update neighbor ordering: $\prec_u = \prec_u \cup U_u$
    5: }

 II Broadcast $\prec_u$ to each node $v \in \prec_u$:
    1: for (each node $v \in \prec_u$) {
    2:     if ($\prec_v$ not yet received) {
    3:         establish connection to node $v$
    4:         send $\prec_u$ to node $v$
    5:     }
    6: }

III Select topology control neighbors:
    1: for (each received order $\prec_v$) {
    2:     if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$) disconnect
    3: }
```

Figure 3.4: Realistic formulation of the XTC algorithm, considering neighbor discovery

Therefore, terminating Step I) to early may result in an incomplete neighbor ordering, which in turn may result in a neighbor node in G_0 that is never going to receive the order established by u. Thus, the resulting topology control graph G'_{XTC} may contain not all links that should be part of G_{XTC} due to an insufficient neighbor discovery phase. In a worst case scenario this may result in a partitioned network.

3.3 Iterative Network Establishment

In this section we're going to present another variant of the XTC algorithm, which does not include a stop condition for the neighbor discovery process. Thus, the devices are infinitely looking for new neighbor devices, and as soon as a new device appears it gets integrated into the existing topology control graph G_{XTC}. By choosing this approach, the problem of undiscovered neighbor nodes gets solved.

To derive this variant of the XTC algorithm, we simply combine steps I) and II) of the XTC definition presented in figure 3.4 as follows:

```
I Broadcast $\prec_u$ to $u$'s neighbors in $G_0$:
1:  while (true) {
2:      discover neighbor devices: $D_u$
3:      $U_u := \{w \in D_u : w \notin \prec_u\}$
4:      update neighbor ordering: $\prec_u := \prec_u \cup U_u$
5:      for (each node $v \in U_u$) {
6:          if ($\prec_v$ not yet received) {
7:              establish connection to node $v$
8:              send $\prec_u$ to node $v$
9:          }
10:     }
11: }
```

3.3.1 Incomplete neighbor order

Let's consider the situation after a node u has done Step I) of the modified XTC algorithm shown above once, that is, node u has sent its order to each neighbor discovered in Line 1 of Step I). After the neighbor nodes have processed the neighbor order $\prec_u$ received from node u, a connection to each node v in $\prec_u$ either exists or not, depending on the decisions made by the neighbor nodes that received $\prec_u$.

Assume now, that node u will have discovered all of its neighbor devices after executing the while loop shown above twice, but won't have discovered all of them after executing Lines 2-8 once. Thus, when executing the while loop for the second time, node u discovers new nodes that were not contained in $\prec_u$ (Line 2), i.e. the set U_u won't empty (Line 3). Node u then updates its neighbor order $\prec_u$ by adding each v in U_u to it (Line 4), establishes a connection to the nodes in U_u (Line 7), and sends its updated neighbor order $\prec_u$ to them (Line 8).

It's clear that the XTC algorithm gets applied correctly to the nodes that were in U_u (that is they get integrated correctly into the existing network). But since the neighbor order $\prec_u$ has changed after executing Line 4 for the second time, it is obvious that the neighbor order sent in the first step was not complete, i.e. it did possibly not contain all the information needed to let its neighbors do a "correct" decision. Thus, the resulting topology control graph G'_{XTC} may differ from the desired topology control graph G_{XTC}.

Let's take a look at what steps have to be taken by node u for that "wrong" decisions can be corrected.

3.3.2 Unconnected Nodes

Let's consider a node v in $\prec_u$ that had no connection to u after processing the while loop once. We now that

$$(u,v) \notin G'_{XTC}, \quad \text{if } (S'_{v,\prec_u} \cap S'_{u,\prec_v} \neq \emptyset),$$

that is, a better common node $b_{u,v}$ exists. After executing Line 4 for the second time, the new devices discovered on Line 2 (Line 3 resp) were added to the neighbor order $\prec_u$. But since the order $\prec_u$ only grows by adding some new neighbor device x to it, the set $S_{v,\prec_u} \cap S_{u,\prec_v}$ will still be not empty. This is shown in the table below, where x is some newly discovered node:

$$\begin{array}{l|c|l}
\boldsymbol{\prec_u} \quad \overbrace{\cdots\ b_{u,v}\ \cdots\ x\ \cdots}^{S_{v,\prec_u}} & \text{v} & \cdots \\
\hline
& \left.\begin{array}{c} \vdots \\ b_{u,v} \\ \vdots \end{array}\right. & \left.\vphantom{\begin{array}{c} \vdots \\ b \\ \vdots \end{array}}\right\} S_{u,\prec_v} \\
& \mathbf{u} & \\
& \vdots &
\end{array}$$

Informally speaking, discovering new neighbor nodes does not change the fact that at least one better common neighbor $b_{u,v}$ exists which is the reason for that the link (u,v) is not in G'_{XTC}. Hence, if a node u discovers new neighbor devices, there's no need for establishing new connections.

3.3.3 Connected Nodes

On the other side, we know that for each node v in $\prec_u$ that is connected to u (i.e. $(u,v) \in G'_{XTC}$) the following condition holds:

$$(u,v) \in G'_{XTC}, \quad \text{if } (S'_{v,\prec_u} \cap S'_{u,\prec_v} = \emptyset).$$

In this case, it might be possible that a node u discovers a new node x that is "better" than a node v to which a connection exists (that is, x gets added to the order $\prec_u$ before v), and hence the node x gets added to the set $S_{v,\prec_u}$.

This may cause the set $S_{v,\prec_u} \cap S_{u,\prec_v}$ to be non-empty, and therefore will cause the link (u, v) to be not in G_{XTC}. This situation is shown in the table below, where x is a newly discovered neighbor:

$$\begin{array}{r|c|l}
\prec_{\boldsymbol{u}}\quad \overbrace{\cdots\ b_{u,v} := x\ \cdots}^{S_{v,\prec_u}} & \text{v} & \cdots \\ \hline
 & \begin{array}{c}\vdots\\ \text{x}\end{array} & \left.\vphantom{\begin{array}{c}\vdots\\ x\end{array}}\right\} S_{u,\prec_v} \\
 & \vdots & \\
 & \text{u} & \\
 & \vdots &
\end{array}$$

But since the link (u, v) was in G'_{XTC} after processing the while loop once, a connection between u and v exists, thus the only thing that has to be done by node u is to re-send its updated neighbor order $\prec_u$ to node v. Node v then processes the updated neighbor order $\prec_u$ again, and terminates the connection if necessary, that is if x actually became a better common neighbor $b_{u,v}$ (see observations 2.5.1, 2.5.2 resp).

Informally speaking, "wrong decisions" due to an incomplete neighbor order can be corrected by broadcasting the updated order to *already connected* neighbors. Hence, a node u correctly integrates newly discovered devices into the (partial) topology control graph if the XTC algorithm is executed in the following way:

```
I Broadcast $\prec_u$ to neighbor devices:
 1: while (true) {
 2:    discover neighbor devices: $D_u$
 3:    $U_u := \{w \in D_u : w \notin \prec_u\}$
 4:    update neighbor order: $\prec_u := \prec_u \cup U_u$
 5:    for (each node $v \in U_u$) {
 6:        if ($\prec_v$ not yet received) {
 7:            establish connection to node $v$
 8:        }
 9:    }
10:    broadcast $\prec_u$ to each connected neighbor
11: }
```

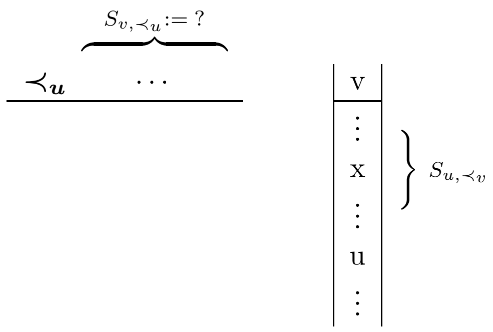
3.3.4 Passive Neighbor Discovery

In the last sections, we derived a variant of the XTC algorithm that eliminates the problem of undiscovered neighbor nodes: since the nodes are looking for new neighbor devices infinitely, each node will discover each of its neighbors that are actually present after some (infinite) amount of time.

In this section, we're going to take a look at another problem that may arise because of insufficient neighbor discovery. More precisely we're going to take a look at the following conflict:

- Node v discovers node u but u does not see v

The problem is obvious: since v discovered u, node u is some when going to receive the neighbor order $\prec_v$ established by node v. To decide if the link (u, v) has to be part of the topology control graph or not, node u has to build the sets $S_{v,\prec_u}$ and $S_{u,\prec_v}$. But since u has not yet discovered node v, node u won't be able to build the set $S_{v,\prec_u}$ and is thus not able to decide if the link should be kept or not:



Basically, two different solutions exist for solving this problem:

Neighbor Ignoring Node u terminates the connection since it cannot find node v in its neighbor order. At a later date, node u will discover node v and the correct decision may be made then by node v.

Passive Neighbor Discovery Node u adds v to its neighbor order, and decides immediately if the link should be kept or not.

The advantage of passive neighbor discovery compared to the first solution is obvious: the link (u, v) gets established only once, hence by choosing the second solution a connection establishment can be saved.

The drawback of passive neighbor discovery becomes clear if we write out this solution in detail, i.e. if we modify the link selection process of the XTC algorithm (Step III):

```
III Select topology control neighbors:
    1: for (each received order $\prec_v$) {
    2:     if ($v \notin \prec_u$) {
    3:         update neighbor order: $\prec_u = \prec_u \cup v$
    4:         broadcast $\prec_u$ to all neighbors
    5:     }
    6:     if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$) disconnect
    7: }
```

Since the unknown device v gets added to the neighbor order $\prec_u$ (Line 3), node u has to inform its neighbors about the changes occurred by sending the updated order to each *connected* neighbor, as explained in the previous subsection. Thus, the order sent by node v may cause a broadcasting of the neighbor order $\prec_u$, which may cause a lot of traffic (depending on the number of neighbors that are connected to node u).

Nevertheless we choose the second solution for solving the conflict mentioned above because the second solution comes along with another interesting advantage that becomes clear if we summarize the results obtained so far, and write the XTC algorithm in its iterative form as derived during this section. This is shown in figure 3.5.

Suppose that a network has just been established by running the iterative XTC algorithm depicted in figure 3.5 on each network node. After some amount of time, the resulting topology control graph G'_{XTC} is equal to the topology control graph G_{XTC}. Further assume that there's an outstanding observer who recognizes that G'_{XTC} equals G_{XTC} and thus forces the nodes to stop looking for new neighbor devices, i.e. they won't execute the while loop in Step I) of the iterative XTC algorithm shown in figure 3.5 anymore.

It's clear, that if a new node x now gets added to the existing network, the network would not be able to integrate the node correctly to the existing network if we do not make use of passive neighbor discovery since the nodes of the existing network do not look for new neighbor devices anymore.

Anyhow, by letting the nodes in the existing network passively discover node x, proper integration into an existing network becomes possible. In the next subsection we're going to explain why this behavior is desired.

```
XTC Algorithm

I) Broadcast $\prec_u$ to neighbor devices:
 1:  while (true) {
 2:      discover neighbor devices: $D_u$
 3:      $U_u := \{w \in D_u : w \notin \prec_u\}$
 4:      update neighbor order: $\prec_u := \prec_u \cup U_u$
 5:      for (each new node $w \in U_u$) {
 6:          if ($\prec_w$ not yet received) {
 7:              establish connection to node $w$
 8:          }
 9:      }
10:      broadcast $\prec_u$ to all connected neighbors
11:  }

II) Select topology control neighbors:
 1:  for (each received order $\prec_v$) {
 2:      if ($v \notin \prec_u$) {
 3:          update neighbor order: $\prec_u = \prec_u \cup v$
 4:          broadcast $\prec_u$ to all neighbors
 5:      }
 6:      if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$) disconnect
 7:  }
```

Figure 3.5: XTC: Iterative formulation

3.4 Conclusion

In this section, we've derived a variant of the XTC algorithm that does neighborhood discovery and connection establishments in an iterative fashion. Furthermore, the iterative XTC algorithm ensures that the resulting topology control graph G'_{XTC} converges to the XTC topology control graph G_{XTC} after some (infinite) amount of time (depicted in figure 3.5. The guarantee that each node discovers all its neighbor devices and thus the resulting topology control graph equals G_{XTC} was gained gained by letting the nodes repeatedly scan there neighborhood.

In practice, implementing the iterative XTC algorithm straight forward results in a power consuming implementation, because neighbor discovery is a very expensive operation. Therefore, it is desirable to advice the nodes to stop executing the while loop (Step I, figure 3.5) of the XTC algorithm - or at least to reduce its frequency - after the topology control graph equals G_{XTC}, and let the nodes only passively discover new devices (as explained in the previous subsection), i.e. the nodes shall still run Step II) of the iterative algorithm (figure 3.5). Hence, we still need some condition that switches the network nodes from this “active” state (executing Step II) into a more “passive” state (i.e. stop executing Step I), or at least reducing the frequency of executing Step I).

Therefore, the question arises what we've gained with the iterative version of the XTC algorithm derived in this section, compared to the version introduced in the last section (figure 3.4). The following list summarizes the most important advantages of the XTC algorithm in its iterative formulation:

Continuous Mode Change The iterative XTC algorithm allows doing a “smooth” transition from neighbor discovery mode to non-discovery mode. This can be reached by continuously reducing the frequency of the while loop (Step I).

Background Neighbor Discovery Process Instead of turning neighbor discovery mode completely off, it is possible to let the neighbor discovery process run at a very low frequency after some predefined amount of time. This can in fact be a good compromise to guarantee that the resulting topology control graph converges to G_{XTC} with reasonable power consumption.

Passive Neighbor Discovery Passive neighbor discovery enables the network to discover new nodes and integrating them correctly into the network even if the nodes of the existing network are not looking for new neighbor devices. Of course we could enable passive neighbor discovery also for the algorithm shown in figure 3.4, but it is obvious that the probability of adding the node correctly to the network will be lower, since the probability of detecting all neighbor devices usually is lower.

Controllability In practice, the assumption of an external observer is not that inappropriate (e.g. observing the network by connecting one of the nodes to a computer). The iterative variant of the XTC algorithm can easily be extended to let an external observer turn on/off neighbor discovering after the network has converged to G_{XTC}. This is useless in an implementation that uses a fixed number of neighbor discovery steps.

Continuous Network Adjustment Another problem of the XTC algorithm presented in figure 3.4 arises if the the network nodes are turned on asynchronously: some nodes may still be in neighbor discovery mode, while other nodes already switched to connection establishment mode, thus the resulting topology control graph usually will contain more links than the desired topology control graph G_{XTC} and has to be “cleaned up” at some time (as explained in section 3.3.3) too. In its iterative variant however, the XTC algorithm does this “clean up” procedure after each neighbor discovery step, i.e. the “mistakes” done are corrected continuously.

Connectivity Since the iterative XTC algorithm starts establishing the first connections right after each neighbor discovery process, connectivity gets usually achieved earlier.

Neighborhood Changes Consider some node u that moves from its initial neighborhood into a completely different neighborhood, i.e. node u is brought to an environment consisting of nodes not yet discovered by node u. It’s obvious that a node running the XTC algorithm presented in figure 3.4 is not able to react upon this situation, while the iterative XTC algorithm is at least able to discover the unknown nodes of its new surrounding. We’re going to discuss this in more detail in section 4.6.2

Chapter 4

Adaptive XTC

In this chapter we're going to derive a variant of the XTC algorithm that is able to adapt the network topology dynamically if the weights of the edges in G are changing. The target was to obtain a network that is able to adapt its topology if its nodes are re-arranged. Thus, we assumed that the network is static most of the time, and changes in the link weights occur infrequently. Section 4.1 explains the problem task in more detail, while in section 4.2 the basics of link weight updating is discussed. In section 4.3, a rough draft of the approach chosen is given. This rough draft will be improved during sections 4.4 and 4.5. A summery of the adaptive XTC algorithm obtained is given in figure 4.12 and figure 4.13.

In our approach, link weight updating is not synchronized, thus we introduce asymmetry in the network. The problems of asymmetry and the countermeasures taken are presented in section 4.6.

Last but not least, the network should be able to deal with broken links and leaving (malfunctioning) devices (e.g. due to power-down). Hence, link and device losses are discussed in section 4.7.

4.1 Adjusting the Network to G^*_{XTC}

Basically, if the node constellation changes, that is G changes, the topology control graph G_{XTC} may change as well, hence the topology control graph established has to be updated. More precisely, if the topology graph G changes to G^*, the topology control graph G_{XTC} has to be adapted to the topology control graph G^*_{XTC} that would be obtained by applying the XTC algorithm on G^*.

But, given that each node u has its neighbor order up to date[1], updating the existing network G_{XTC} to G^*_{XTC} is simple: Since each node always "listens" for neighbor orderings from neighbor devices, the only thing that has to be done is to exchange the current neighbor orderings among the nodes. A receiving node v may then process the received order $\prec_u$ and decide if the link (u, v) has to be closed, if it has to be established, or if it simply has to be kept.

But since not all neighbors are connected directly, there are only two possibilities to guarantee that each neighbor is going to receive the order of each of its neighbors:

Broadcasting Each node u periodically broadcasts its updated neighbor ordering to *all* nodes in the network. A receiving node v may then decide if the link (u, v) has to be established / closed.

Connection trials Each node u periodically establishes connections to its unconnected neighbors and sends its updated neighbor order $\prec_u$ to them. A receiving node v may then decide if the link (u, v) should be kept or closed.

The main problems of both solutions are obvious: The second solution results in a lot of unnecessary connection establishments, especially if we assume that G changes slowly. On the other side, by periodically broadcasting neighbor orders, unnecessary connection establishments can be avoided, but the network gets flooded with a lot of unnecessary packets. Furthermore, controlling broadcast packets in cyclic networks needs a lot of effort. Thus, none of the two solutions is efficient in its rough draft.

Anyhow, this thesis focuses on the second solution mentioned above. The main reason is that a node u may prevent a lot of unnecessary connection establishments by simply observing changes in its neighbor ordering $\prec_u$ resulting from a change in the topology of G. How a node u has to observe its order will be part of the following sections. After elaborating this solution we're going to give additional reasons for the choice made here at the end of this section.

[1]i.e. each node immediately adjusts its neighbor order according to the new topology of G^*

4.2 Getting the Link Quality

To update the neighbor order $\prec_u$, a node u has to update the link qualities of all its discovered neighbors in $\prec_u$. Getting the link quality of an edge (u, v) in G_{XTC} is simple: it can be read directly from the existing connection. Anyhow, a node u may obtain the link quality of a link (u, v) that is not in G_{XTC} (i.e. a connection to node v does not exist) only in two different ways[2]:

- by establishing a connection to the neighbor node v and reading the link quality,
- or by neighbor discovering, a node u obtains the weights of all edges to the discovered neighbor device.

This thesis focuses on updating the neighbor order by neighbor discovering for the following reasons:

1. Updating the neighbor order $\prec_u$ after each neighbor discovery step can be integrated easily into the iterative XTC algorithm introduced in section 3.3 in figure 3.5.
2. As mentioned in section 3.4 we do not come around continuous neighbor discovering for that a node u is able to integrate itself properly into a new surrounding (i.e. it has to be able to add new neighbors to its order). Furthermore, letting the discovering process (Step I in figure 3.5) run in background at a low frequency increases the probability of integrating new nodes correctly into the existing network. Thus, we can now use this background process additionally for updating the neighbor ordering.
3. Using a low frequency for updating the neighbor ordering should be sufficient, as well as efficient with regards to the assumption that the topology of G_0 is quasi-static.
4. With Bluetooth, connection establishment (paging) and device discovery (inquiry) are resembling processes with regards to power-consumption. But since a single inquiry yields the link weight of all discovered devices, it seems to be more efficient to do a single inquiry than to page each neighbor device.

Necessity of updating unconnected Links

No matter which of the above mentioned strategy gets chosen, updating the link quality of unconnected nodes is a power consuming process: Neighbor

[2]In fact, this is hardware dependent, hence this statement holds for the BTnodes used in this thesis, but may not hold in other systems

discovering, as well as connection establishment (i.e. paging in Bluetooth) are time consuming and costly processes. Therefore, the question arises if it is not possible to come along without updating the quality of unconnected links (e.g. by observing links in G^*_{XTC} only).

The answer is no, as can be seen from the example depicted in figure 4.1: In figure 4.1 a) a network established with the XTC algorithm is shown. The edge (a, c) is not in G_{XTC} because there exists the better common node $b_{a,c} := b$.

Figure 4.1 b) shows what happens if an obstacle gets placed between nodes b and c: The link weight ω_{bc} increases (in Bluetooth: the RSSI at node c decreases), until the link weight ω_{bc} becomes worse than the weight of the edge (a, c). Thus, from the point of view of node c, node a will appear before b in its neighbor ordering $\prec_c$, which can be observed by continuously reading the link quality of the link (b, c). This change in the neighbor ordering $\prec_c$ causes the link (a, c) to be in G_{XTC}, as well as the removal of (b, c) from G_{XTC}.

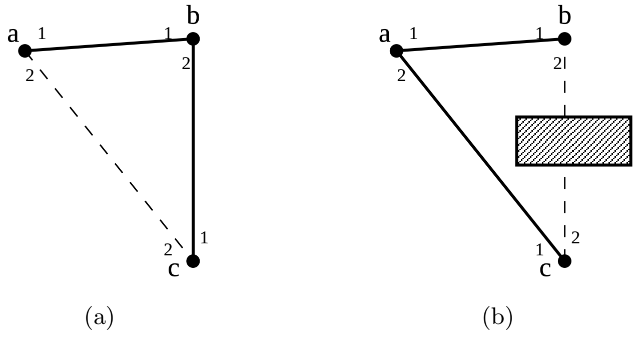


Figure 4.1: Link closing due to an obstacle

Note that until now, there was no need for node c or node d to update the weight ω_{ac} of the edge (a, c) since it did not change. But after removal of the obstacle, the weight of (a, c) will regain its initial value. But since neither node c nor node d is going to observe it, none of the involved nodes is able to bring back the topology control graph to its initial state.

4.3 Approach

We are now going to present the approach chosen in this thesis to derive an adaptive variant of the XTC topology control algorithm. As discussed in the previous section, a node u running the algorithm will adapt the (partially) established topology control graph G_{XTC} to changes in the topology of G after each neighbor discovery.

A node u adapts to the topology control graph G^*_{XTC} by updating its neighbor order $\prec_u$ and by reacting accordingly to the changes observed. More precisely, by observing changes in its neighbor order $\prec_u$ a node u will evaluate:

1. *new candidates*: edges (u, v) that are not in G_{XTC} but that come into question to be in G^*_{XTC}, and
2. *outdated candidates*: established links (u, v) that come into question to be not in G^*_{XTC} (i.e. that have to be closed).

Afterwards, the node u will establish a connection *to each new candidate* and send its updated order $\prec^*_u$ to it. Likewise, node u will send its updated order $\prec^*_u$ to each evaluated *outdated candidate.*

4.3.1 Rough Draft

A rough draft of this solution is given in figure 4.2: Before starting a new neighbor discovery step (Line 4), a copy of the current neighbor order $\prec^*_u$ is established (Line 3), and the sets C_u and R_u are initialized to be empty (Line 2).

```
XTC Algorithm

I) Broadcast $\prec_u$ to neighbor devices:
 1: while (true) {
 2:     $R_u := \{\}$, $C_u := \{\}$
 3:     copy neighbor ordering: $\prec_u := \prec^*_u$
 4:     discover neighbor devices: $D_u$
 5:     for (each node $v \in D_u$) {
 6:         if ($v \in \prec_u$) {
 7:             update neighbor order $\prec^*_u$ according to $\omega^*_{uv}$
 8:         }
 9:         else $\prec^*_u := \prec^*_u \cup v\,(id_v, \omega_{uv})$
10:     }
11:     $C_u :=$ get_new_candidates$(\prec^*_u, \prec_u)$
12:     $R_u :=$ get_outdated_candidates$(\prec^*_u, \prec_u)$
13:     apply_changes$(R_u, C_u)$
14: }
```

Figure 4.2: Adaptive XTC: Approach

From Lines 5-10 the current neighbor order $\prec^*_u$ gets updated according to the neighbor discovery result D_u obtained in Line 4: if the node v can be found in the outdated neighbor order $\prec_u$, the neighbor order gets updated according to the new weight ω^*_{uv} of the edge (u, v) found during neighbor discovery (Line 7). If a node v in D_u was not yet known (i.e. v cannot be found in $\prec_u$), the node gets added to the neighbor order, that is its Id and the weight of the edge ω_{uv} are stored in the neighbor order (Line 9).

Afterwards, the updated neighbor order $\prec^*_u$ gets compared with the copy of the initial order $\prec_u$: On Line 11 the set C_u is established that contains the new candidates evaluated. Outdated candidates are evaluated and added to the set R_u at Line 12.

Finally, the node u reacts upon the changes observed by sending its updated order $\prec^*_u$ to each candidate found on Lines 11-12, that is node u will establish a connection to each v in C_u and send its updated order to it. Likewise, node u will send its updated order $\prec^*_u$ to each node contained in the set R_u.

Note that Step II) needs not be adjusted for the moment.

4.3.2 Targets

As mentioned before, the main target of the adaptive formulation of the XTC algorithm is that the current topology control graph G_{XTC} converges to G^*_{XTC}. This will certainly be the case if *all* edges (u, v) adjacent to u are "checked" after each update of the neighbor order, i.e. if all unconnected nodes are considered as new candidates, and all connected nodes are considered as outdated candidates. This can be reached by implementing the functions *get_new_candidates*() and *get_outdated_candidates*() as shown in figure 4.3.

```
XTC Algorithm

get_new_candidates($\prec^*_u, \prec_u$):
1:  $C := \{\}$
2:  for (each unconnected node $v \in \prec^*_u$) {
3:      $C := C \cup v$
4:  }
5:  return $C$

get_outdated_candidates($\prec^*_u, \prec_u$):
1:  $R := \{\}$
2:  for (each connected node $v \in \prec^*_u$) {
3:      $R := R \cup v$
4:  }
5:  return $R$
```

Figure 4.3: Adaptive XTC: Unoptimized candidate evaluation

Secondly, the adaptive XTC algorithm should be efficient with regards to power consumption. Mainly, this can be reached by reducing unnecessary node activity, that is

- saving unnecessary connection establishments,
- and reduce excessive order exchange.

In other words, we are interested in minimizing the sets of candidates (i.e. the sets R_u and C_u in figure 4.2) before calling the *apply_changes*() by optimizing the functions *get_new_candidates*() and *get_outdated_candidates*(). Thus, in an optimal scenario, the set R_u will only contain the links (u, v) that actually have to be closed, i.e.

$$R_u := \{v \in \prec_u^*: (u, v) \in G_{XTC} \wedge (u, v) \notin G^*_{XTC}\},$$

whereas the set C_u will only contain the links (u, v) that actually have to be established, i.e.

$$C_u := \{v \in \prec_u^*: (u, v) \notin G_{XTC} \wedge (u, v) \in G^*_{XTC}\}.$$

4.4 Optimization by Exclusion

In this section we're going to present how neighbor nodes in $\prec_u$ can be excluded from both, the set of new candidates, as well as of the set of outdated candidates with certainty. This can be reached by a simple comparison of the neighbor order $\prec_u$ before the update occurred and the updated neighbor order $\prec_u^*$.

4.4.1 Excluding connected neighbors

Let's start with the simpler case: a node v in $\prec_u$ to which a connection exists (i.e. $(u, v) \in G_{XTC}$). Since the link (u, v) is in G_{XTC}, we know that the set $S_{v,\prec_u} \cap S_{u,\prec_v}$ is empty before updating the neighbor order $\prec_u$, i.e. no better common neighbor exists. Thus, in tabular representation, the situation before updating the neighbor order $\prec_u$ looks like this:

$$\begin{array}{lc|c|l}
\boldsymbol{\prec_u} & \overbrace{\quad\cdots\quad}^{S_{v,\prec_u}} & \text{v} & \cdots\ \text{x}\ \cdots \\ \hline
 & & \vdots & \\
 & & \text{x} & \Big\} \; S_{u,\prec_v} \\
 & & \vdots & \\
 & & \text{u} & \\
 & & \vdots &
\end{array}$$

where node x indicates that common neighbors may exist, but none of these neighbors is a better common neighbor b_{uv} since $(u, v) \in G_{XTC}$.

Assume now, that after updating the neighbor order $\prec_u$, node x appears before v in the updated neighbor order $\prec_u^*$. Given that the neighbor order of v did not change, and the link selection decision would be repeated, the link (u, v) would have to be removed from G_{XTC}, since the set $S^*_{v,\prec_u} \cap S_{u,\prec_v}$ would now contain x, thus would not be empty anymore. This can be seen from the tabular representation shown below:

$$\begin{array}{rc|c|cl}
\boldsymbol{\prec_u^*} & \overbrace{\cdots \;\; \text{x} \;\; \cdots}^{S^*_{v,\prec_u}} & \text{v} & \cdots & \\
\hline
 & & \vdots & & \\
 & & \text{x} & \left.\vphantom{\begin{array}{c}\vdots\\ \text{x}\\ \vdots\end{array}}\right\} & S_{u,\prec_v} \\
 & & \vdots & & \\
 & & \text{u} & & \\
 & & \vdots & &
\end{array}$$

Hence, if some node x appears before a *connected* node v in the updated order $\prec_u^*$ that did not appear before v in $\prec_u$, it may be possible that the link (u, v) has to be closed, i.e. the *connected* node v is an outdated candidate. Thus

Observation 4.4.1 *A connected node v in $\prec_u^*$ is an outdated candidate if and only if*

$$\exists x \in S^*_{v,\prec_u} : x \notin S_{v,\prec_u}.$$

4.4.2 Excluding unconnected neighbors

We're now going to take a look at nodes v to which some node u has no connection, i.e. the set $S_{v,\prec_u} \cap S_{u,\prec_v}$ is non-empty. Hence, at least one common better neighbor b_{uv} exists before updating the neighbor order:

$$\begin{array}{rc|c|cl}
\boldsymbol{\prec_u^*} & \overbrace{\cdots \;\; b_{uv} \;\; \cdots}^{S^*_{v,\prec_u}} & \text{v} & \cdots & \\
\hline
 & & \vdots & & \\
 & & b_{uv} & \left.\vphantom{\begin{array}{c}\vdots\\ \text{x}\\ \vdots\end{array}}\right\} & S_{u,\prec_v} \\
 & & \vdots & & \\
 & & \text{u} & & \\
 & & \vdots & &
\end{array}$$

Assume that after updating the neighbor order, node v appears before the better common neighbor b_{uv} in $\prec_u^*$ thus, given that b_{uv} was the only better common node of the link (u, v), the link (u, v) has to be added to G_{XTC} thus in this case, a connection to node v has to be established.

Thus, whenever some node x appears after an *unconnected* node v in the updated order $\prec_u^*$ but appeared before v in $\prec_u$, it may be that x was a better common neighbor b_{uv} of the edge (u, v) and hence the link (u, v) has to be established, i.e. the neighbor v is a new candidate. In other words:

Observation 4.4.2 *A unconnected node v in $\prec_u^*$ becomes a candidate if and only if*

$$\exists x \in S_{v,\prec_u} : x \notin S^*_{v,\prec_u}.$$

4.4.3 Conclusion

Integrating the ideas of exclusion into our formulation of the adaptive XTC algorithm presented in figure 4.2 to reduce node activity can now be done by writing out the functions *get_new_candidates*() and *get_outdated_candidates*() in more detail as shown in figure 4.4.

XTC Algorithm

get_new_candidates$(\prec_u^*, \prec_u)$:
```
1: $C := \{\}$
2: for (each unconnected node $v \in \prec_u^*$) {
3:     if $(v \notin \prec_u)$   $C := C \cup v$
4:     else if $(\exists x \in S_{v,\prec_u} : x \notin S^*_{v,\prec_u})$   $C := C \cup v$
5: }
6: return $C$
```

get_outdated_candidates$(\prec_u^*, \prec_u)$:
```
1: $R := \{\}$
2: for (each connected node $v \in \prec_u^*$) {
3:     if $(\exists x \in S^*_{v,\prec_u} : x \notin S_{v,\prec_u})$   $R := R \cup v$
4: }
5: return $R$
```

Figure 4.4: Network maintenance: Reducing node activity by exclusion

It's clear that the optimized evaluation functions presented in figure 4.4 reduce node activity compared to the evaluation functions presented in figure 4.3. As an example consider the network depicted in figure 4.5. Assume that the position of node b changes as indicated in the figure. If the nodes would make use of the unoptimized evaluation functions shown in figure 4.2, node a would establish a connection to both, node y and node c after the next neighbor discovery operation has been executed.

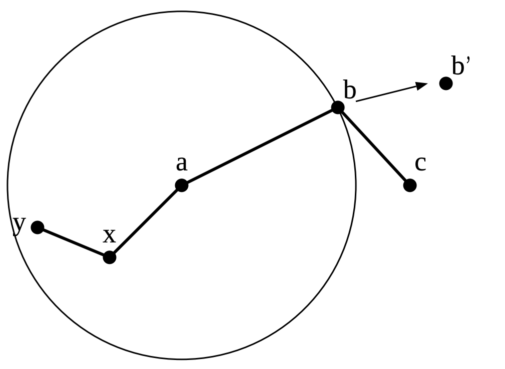


Figure 4.5: XTC maintenance with exclusion: an example

But by applying the principle of exclusion (i.e. the nodes make use of the optimized evaluation functions presented in figure 4.4), the unnecessary connection establishment to node y gets saved, since the set $S_{y,\prec_a}$ does not change when the position of b changes.

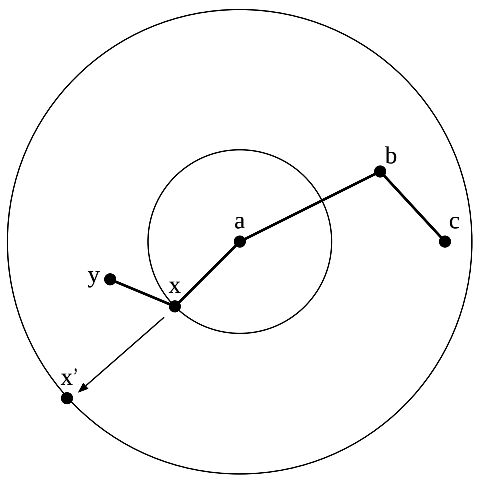


Figure 4.6: XTC maintenance with exclusion: second example

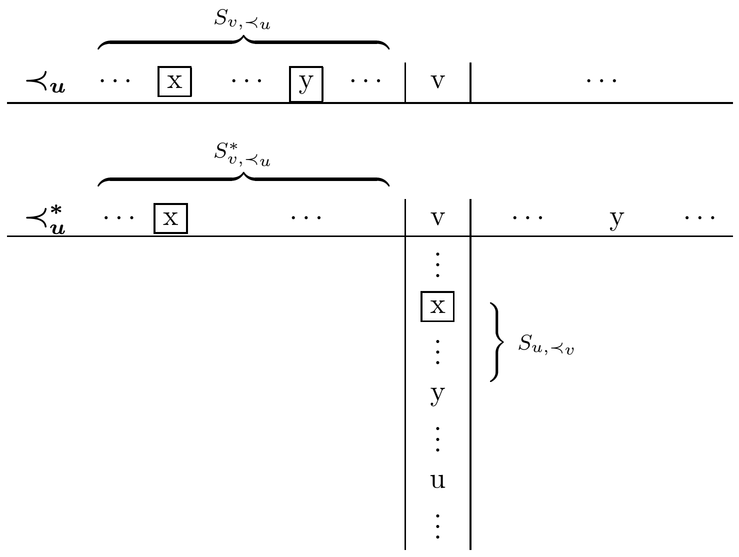
Nevertheless, preserving unnecessary connection establishments by exclusion is a matter of luck and depends strongly on the actual situation and the occurring topology changes. This is shown in figure 4.6, that shows the same network as in the previous example (figure 4.5) but a different change in topology. Since x is nearest to node a, node x appears before node y, as well as before c in $\prec_u$. After the change in topology occurred (that is the position of x changed), both, node y, as well as node c will appear before x. Hence node a will establish the links (a, y) and (a, c) because x cannot be found in the set $S^*_{y,\prec_a}$, as well as in the set $S^*_{c,\prec_a}$. Informally speaking, node a will establish the links (a, y) and (a, c) because node x may *possibly* have been a common better neighbor.

You may have noticed that in the discussion above we only considered how unnecessary connection establishments can be saved. This is because unnecessary connection establishments is the more severe problem than excessive order exchange: the expense of a connection establishment is several times higher than the cost of sending a packet to a connected neighbor device. Thus, although the exclusion principle does not really decrease excessive order exchange considerably, we are not going to make any efforts to reduce it further.

4.5 Better Common Neighbors

We are now going to introduce a concept that allows a node u to reduce the set of new candidates C_u further, and hence to preserve a lot of unnecessary connection establishments. This can be obtained by extending the principle of exclusion of unconnected neighbors.

The problem of the exclusion principle presented in section 4.4 is that a node u has to establish a connection each time a neighbor may *possibly* be a common better neighbor b_{uv}, i.e. every time a node x disappears from the set $S_{v,\prec_u}$ an unnecessary connection establishment occurs if at least one different common better neighbor exists. An example is given below, where x and y are both better common neighbors, thus the disappearance of one of them from the set $S_{v,\prec_u}$ (in this case y) does not cause an inclusion of (u, v) in G^*_{XTC}.


$S_{v,\prec_u}$
$\prec_u$ ··· x ··· y ··· v ···
$S^*_{v,\prec_u}$
$\prec^*_u$ ··· x ··· v ··· y ···
x
y
u
$S_{u,\prec_v}$

From the table above it becomes clear that a connection to some unconnected neighbor v actually has to be established (i.e. if $(u, v) \in G^*_{XTC}$) not before *all* common better neighbors disappeared from $S_{u,\prec_v}$. Hence, if an unconnected pair (u, v) focuses only on one of them, a node u can save a lot of unnecessary connection establishments.

4.5.1 Concept

The idea is quite simple: when a node v receives a neighbor order $\prec_u$ and v decides to close the connection, the nodes u and v agree upon one of the common better neighbors. Instead of just storing the pair $\{id_u, \omega_{uv}\}$ in the neighbor ordering $\prec_v$, node v additionally stores the better common neighbor b_{uv} chosen, i.e. node v stores the triple $\{id_u, \omega_{uv}, b_{uv}$ in its neighbor ordering. Node v will then not establish the link (u, v) until the better common node b_{uv} has disappeared from the set $S_{u,\prec_v}$. In other words: if the edge (u, v) is not in G_{XTC}, node v will not establish the link until node u appears before b_{uv} in its updated neighbor order $\prec_v^*$.

If also node u stores the triple $\{id_v, \omega_{uv}, b_{uv}\}$ in its neighbor order, and if node u also establishes the link (u, v) not before v appears before b_{uv} in $\prec_v$, unnecessary connection establishments can be saved. Thus, we can introduce the evaluation function shown in figure 4.7 which reduces the set of new candidates C_u by making use of the better common neighbor concept.

```
XTC Algorithm

get_new_candidates($\prec_u^*$):
1:  $C := \{\}$
2:  for (each unconnected node $v \in \prec_u^*$) {
3:      if ($v \prec_u^* b_{uv}$)   $C := C \cup v$
4:  }
5:  return $C$
```

Figure 4.7: Adaptive XTC: improved evaluation function for reducing the set of new candidates

Note that the function shown in figure 4.7 does only depend on the updated neighbor ordering $\prec_u^*$, that is the set C can be established without the knowledge of the initial neighbor order $\prec_u$. To reach this, we have to ensure that also new nodes found during neighbor discovery can be determined in the updated neighbor order. But this is already the case: for a newly discovered edge (u, v) adjacent to u, no better common neighbor b_{uv} has been determined, hence the condition on Line 3 in figure 4.7 will also be true for newly discovered devices.

4.5.2 Example

An example is given in figure 4.8: Assume that nodes a, x, and b do not change their positions, but that node c periodically changes its position as indicated. If the nodes would make use of the evaluation function presented in figure 4.4, node a would unnecessarily establish a connection, each time node c moves to the position c', respectively the link (a, b) would unnecessarily be established by node a each time node c moves from position c' to the position c.

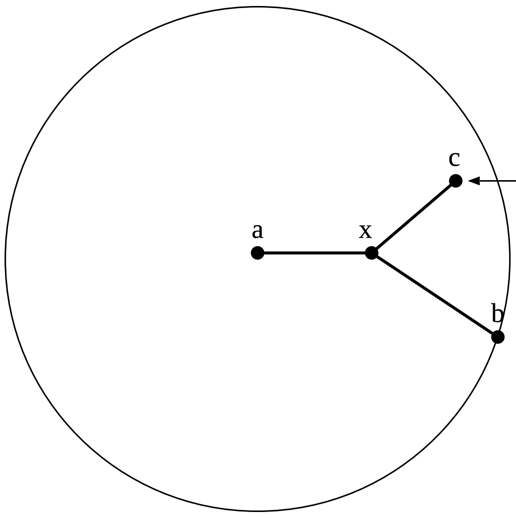


Figure 4.8: XTC maintenance: Saving unnecessary connection establishments by choosing a common better neighbor

But by making use of the evaluation function introduced before (figure 4.7), node x is the better common neighbor for both, link (a, c), as well as for link (a, b). Thus node a does neither establish the link (a, b) nor the link (a, c) if the position of c varies, since neither b nor c appears before the better common neighbor x in $\prec_a$. In the same way it can be shown that also b, as well as c do not establish a connection to node a as the position of c varies. Thus no unnecessary connection will be established in the situation depicted in figure 4.8, as it should be.

4.5.3 Adjusting XTC Step II

For that the sender of the neighbor order $\prec_u$ is able to store the common better neighbor b_{uv}, the receiving node v has to tell node u who is the better common neighbor. Thus, to be able to make use of the concept introduced before, we have to make use of the 2-way handshake protocol depicted in figure 3.3. Hence we have to adjust Step II) in figure 4.2 of the XTC algorithm as shown in figure 4.9.

```
XTC Algorithm

II) Select topology control neighbors:
 1:  for (each received packet) {
 2:      switch (packet type) {
 3:      case (order packet $\prec_v$)
 4:          if ($v \notin \prec_u$) {
 5:              update neighbor order: $\prec_u = \prec_u \cup v$
 6:              broadcast $\prec_u$ to all neighbors
 7:          }
 8:          if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$) {
 9:              choose better common node $b_{uv} \in (S_{v,\prec_u} \cap S_{u,\prec_v})$
10:              store better common node $b_{uv}$ in $\prec_u$
11:              send 'NACK' packet to $v$
12:          }
13:          else {
14:              send 'ACK' packet to $v$
15:              inform application layer
16:          }
17:      case ('NACK' packet)
18:          store better common node $b_{uv}$ in $\prec_u$
19:          disconnect
20:      case ('ACK' packet)
21:          inform application layer
22:      }
23:  }
```

Figure 4.9: Adaptive XTC: making use of better common nodes

Basically, a receiving node u now has to distinguish three different packet types: received neighbor order $\prec_v$, 'ACK' packets, and 'NACK' packets, as indicated in the Lines 2, 3, 17, and 20.

If a neighbor order $\prec_v$ is received, the receiving node u decides if the link (u, v) should be in G_{XTC} or not (Line 8). If the connection has to be closed, the receiving node u has to choose a common better node x from the set $S_{v,\prec_u} \cap S_{u,\prec_v}$ (Line 9), write it to the neighbor order (Line 10), and send back a 'NACK' packet including the Id of the better common neighbor b_{uv} chosen.

If a node u receives a 'NACK' packet, it stores the better common neighbor b_{uv} in its neighbor ordering $\prec_u$, and closes the connection (Lines 18-19). If a 'ACK' packet is received, node u only has to inform the application layer (Lines 20-22), which has also to be done if a node u has decided to keep a connection (Line 15).

In order to enable passive neighbor discovery as explained in section 3.3.4, Lines 4-7 ensure that a received neighbor ordering $\prec_v$ can be processed even if the sending node v has not been discovered so far.

4.6 Asymmetry

In practice, another problem arises because the nodes of an established network do not update their neighbor order synchronously. This leads to asymmetric edges (u, v), that is node u assigns another weight to the link as node v does. In this section, we're going to show that severe problems may arise due to asymmetric links, and what can be done against it.

4.6.1 The Problem

The problem that arises due to asymmetric links can be shown by a simple example, depicted in figure 4.10. Figure 4.10 a) indicates that at some time a change in topology occurs, i.e. the quality of the link (a, c) increases. Assume now, that node c will be the first node that updates its neighbor ordering after the change in topology occurred.

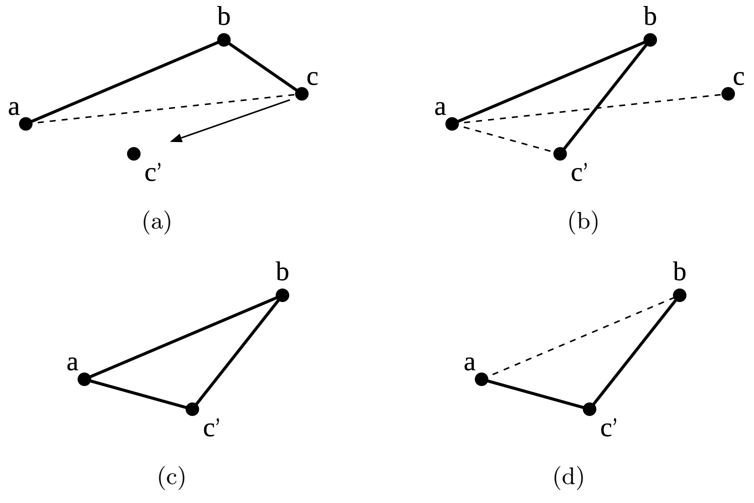


Figure 4.10: Asymmetry: An example

Thus, before updating, the order $\prec_c$ looked like this:

$$\begin{array}{lcc} & & b \\ \prec_c\textbf{:} & \boxed{b} & a \\ \hline \end{array}$$

where the letter b above a indicates that node b is the better common neighbor b_{ac} of the link (a, c), and the box around b indicates that a connection to it exists. After updating, the order $\prec_c^*$ looks like this:

$$\begin{array}{ccc} & b & \\ \boldsymbol{\prec_c^*}\textbf{:} & a & \boxed{b} \\ \hline \end{array}$$

Since node a now appears before the better common neighbor $b_{ac} := b$ of the edge (a, c), node c will establish a connection to node a. Furthermore, node c will send its updated order to node b, since a new node (node a) appears in the set $S^*_{b,\prec_c}$.

After establishing the connection to node a, node c will send its updated order $\prec_c^*$ to node a. Node a will immediately process the received order, and decides if the connection should be kept or not. Since node a has not yet observed the change in topology, the link selection decision in tabular representation will look like this (see figure 4.10 b):

$$\begin{array}{cc|c|} \boldsymbol{\prec_a}\textbf{:} & \boxed{b} & c \\ \hline & & a \\ & & b \end{array}$$

thus node a will decide to keep the link (a, c).

Next, node c will send its updated neighbor order to node b which will not cause the closing of the link (b, c) as can be seen from the tabular representation below:

$$\begin{array}{c|c|c} \boldsymbol{\prec_b}\textbf{:} & \boxed{c} & \boxed{a} \\ \hline & a & \\ & b & \end{array}$$

Thus, after node c has reacted upon the changes in topology, the network will result in the temporary, completely connected network depicted in figure 4.10 c). Thus

Observation 4.6.1 *Theorem 2.3.2 does not hold if the topology graph G consists of asymmetric links.*

It is clear that after node a has observed the change in topology, and adjusted its links as well, the topology control graph will converge to the topology control graph shown in figure 4.10 d). But since the frequency of adapting the network to changes in topology is usually very low in order that power consumption can be reduced to a minimum (as explained in section 3.4), and due to the fact that asynchronous updating cannot be avoided[3], the network may stay a long time in the state shown in figure 4.10 c).

That this may be a severe problem - especially in larger networks - is shown in figure 4.11: Figure 4.11 a) shows an XTC network before the

[3]in fact, asynchronous updating is desired, as we shall see in section 6.1

indicated change in topology occurred, and figure 4.11 b) shows the network after only node c has observed the changes in its neighborhood and adjusted its links accordingly.

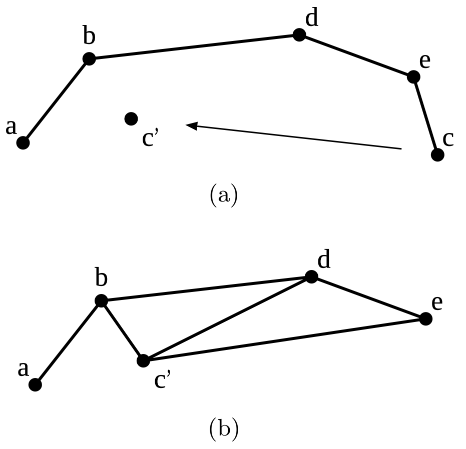


Figure 4.11: Asymmetry: Another example

The problem of a (temporary) topology control graph as depicted in figure 4.11 b) is twofold:

1. Unnecessary power consumption because the excess links are kept longer than necessary.

2. The unnecessary links produce a lot of overhead for the transport layer.

4.6.2 Solution

Solving the problem of asymmetry is straight forward: each time some node u updates its neighbor order (i.e. the weights of all adjacent edges (u, v) in $\prec_u$), u has to inform its neighbors about the changes occurred, which can be done by broadcasting the updated neighbor order to each neighbor device in $\prec_u$. The neighbors may then update the link weight ω_{uv} as well.

This ensures that all established links $(u, v) \in G_{XTC}$ will be symmetric, before doing the link selection decision, that is before executing Lines 8-16 of the adaptive XTC algorithm Step II) depicted in figure 4.9.

Asymmetry in edges (u, v) that are not in G_{XTC} are no problem, as long as both parties are not willing to establish the link. But as soon as the link (u, v) is established by either node u or v, asymmetry can easily be eliminated in the same way right after the order packet arrived.

Hence the straight forward way would be to insert the following piece of pseudo-code between Lines 7 and 8 in XTC Step II (figure 4.9:

```
 8:            else {
 9:                read link weight $\omega^*_{uv}$
10:                update $\prec_u$ according to $\omega^*_{uv}$
11:            }
```

But since the neighbor order may change by updating the link weight ω_{uv}, probably some new connections have to be established, or the updated neighbor order has to be sent to another neighbor node. Thus the complete solution will look as summarized in figure 4.12 and figure 4.13.

To save some space, as well as to point out the communities of Step I) and Step II), we introduced the function *update_neighbor_order*() which executes Lines 5-10 in Step I) of the adaptive XTC algorithm presented in figure 4.2. The function is written out in more detail in figure 4.13.

It's now easy to see that adjusting the network to changes in topology is done in two ways:

1. after a neighbor discovery operation (Step I, Lines 4-7)
2. each time a neighbor ordering $\prec_v$ is received (Step II, Lines 4-6)

The only difference is that in Step I) the whole neighbor order gets updated to the current state of the topology graph G, whereas in Step II) only the link weight ω_{uv} of a single neighbor node v gets updated.

Note that instead of calling the *get_outdated_candidates*() function the neighbor order is broadcasted to all neighbors at Line 9 (execution of the broadcast is done in Line 10) in XTC Step I), in comparison with the adaptive XTC algorithm presented before (figure 4.4). As mentioned above, this is necessary to cancel asymmetry.

But on the other hand, in Step II), Line 18 the updated neighbor order is only sent to the neighbors necessary (as explained in section 4.4). Thus, the broadcast to cancel asymmetry, started by some node u will only propagate to nodes that are affected "directly" by the changes observed by node u.

XTC Algorithm

I) Broadcast $\prec_u$ to neighbor devices:

```
while (true) {
    $R_u := \{\}$, $C_u := \{\}$
    copy neighbor ordering: $\prec_u := \prec_u^*$
    discover neighbor devices: $D_u$
    for (each node $v \in D_u$) {
        $update\_neighbor\_order(\prec_u^*, v)$
    }
    $C_u := get\_new\_candidates(\prec_u^*)$
    $R_u :=$ all connected neighbors in $\prec_u$
    $apply\_changes(R_u, C_u)$
}
```

II) Select topology control neighbors:

```
for (each received packet) {
    switch (packet type) {
    case (order packet $\prec_v$)
        $R_u := \{\}$, $C_u := \{\}$
        read link weight $\omega_{uv}^*$
        copy neighbor ordering: $\prec_u := \prec_u^*$
        $update\_neighbor\_order(\prec_u^*, v)$
        if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$) {
            choose better common neighbor $b_{uv} \in (S_{v,\prec_u} \cap S_{u,\prec_v})$
            store better common neighbor $b_{uv}$ in $\prec_u$
            send 'NACK' packet to $v$
        }
        else {
            send 'ACK' packet to $v$
            inform application layer
        }
        $C_u := get\_new\_candidates(\prec_u^*)$
        $R_u := get\_outdated\_candidates(\prec_u^*, \prec_u)$
        $apply\_changes(\prec_u, R_u, C_u)$
    case ('NACK' packet)
        store better common node $b_{u,v}$ in $\prec_u$
        disconnect
    case ('ACK' packet)
        inform application layer
    }
}
```

Figure 4.12: Adaptive XTC, Part A

XTC Algorithm

```
update_neighbor_order($\prec_u, v$):
1: if ($v \in \prec_u$) {
2:     update neighbor order $\prec_u$ according to $\omega_{uv}$
3: }
4: else $\prec_u := \prec_u \cup v\,(id_v, \omega_{uv}, b_{uv} := \emptyset)$

get_new_candidates($\prec_u$):
1: $C := \{\}$
2: for (each unconnected node $v \in \prec_u$) {
3:     else if ($v \prec_u b_{uv}$)   $C := C \cup v$
4: }
5: return $C$

get_outdated_candidates($\prec_u^*, \prec_u$):
1: $R := \{\}$
2: for (each connected node $v \in \prec_u^*$) {
3:     $R := R \cup v$
4: }
5: return $R$

apply_changes($R, C$):
1: for (each $v \in C$) {
2:     establish connection to $v$
3:     send $\prec_u$ to $v$
4: }
5: for (each $v \in R$) {
6:     send $\prec_u$ to $v$
7: }
```

Figure 4.13: Adaptive XTC: functional parts

4.7 Link Losses

This section deals with an important issue in ad hoc networking: connection / device losses. The effects of both, a connection loss, as well as of a device loss are obvious: the remaining topology control graph G^*_{XTC} may be unconnected. This is shown in figure 4.14.

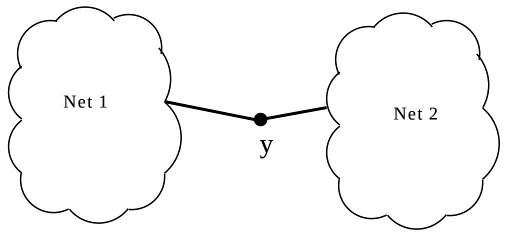


Figure 4.14: Link Loss: Losing node y (e.g. because of a power down) results in an unconnected topology control graph

In this section we're going to take a look at which actions have to be taken by the remaining nodes for that the remaining control graph can be brought to a connected state again.

Connection Losses vs. Device Losses

Usually, the network nodes may not distinguish between a connection loss and a device loss[4], i.e. from point of view of a node u only a link loss is observed.

To distinguish between link losses and device losses the following can be done: after some node u observes a link loss it tries to re-establish the lost link. If reconnection fails, it is assumed that the neighbor device is not able to respond anymore, i.e. will be interpreted as a device loss, that is the device is considered as malfunctioning.

To consider link / device losses, we extend the previous formulation of the XTC algorithm by Step III) as shown in figure 4.15. In case of a connection loss, the observing node u tries to reconnect to the previously connected device y, as mentioned above. If reconnection fails, the link loss is interpreted as a device loss, hence the neighbor device has to be removed from the neighbor order $\prec_u$ (Line 4).

Device Losses

We are now going to take a closer look at the consequences that arise if a node leaves the network. From point of view of the network, the device

[4]at least the BTnode may not

```
XTC Algorithm

III) Link Losses:
1:  for (each lost link $(u, y)$) {
2:      create connection to $y$
3:      if (error) {
4:          remove $y$ from $\prec_u$
5:      }
6:      else send order to $y$
7:  }
```

Figure 4.15: XTC: Link Losses

is "lost", that is it can't be used as a relay node for sending messages to remote devices.

An example of a device loss is given in tabular representation below, where node y is removed from the G, and thus will not appear in $\prec_u^*$ anymore:

$$\boldsymbol{\prec_u} \quad \overbrace{\cdots \quad \boxed{\text{y}} \quad \cdots}^{S_{v,\prec_u}} \quad \text{v} \quad \cdots$$

$$\boldsymbol{\prec_u^*} \quad \overbrace{\qquad\quad \cdots \qquad\quad}^{S^*_{v,\prec_u}} \quad \text{v} \quad \cdots$$

Let's consider if the removal of node y implies that a new connection to some node v in $\prec_u$ has to be established (if no connection to v exists), or if an existing connection to some node v has to be closed.

If a connection to node v exists ($S_{v,\prec_u} \cap S_{u,\prec_v} = \emptyset$)it's obvious: by the removal of a node y the size of the set $S_{v,\prec_u}$ decreases (i.e. $|S^*_{v,\prec_u}| < |S_{v,\prec_u}|$), thus the set $S^*_{v,\prec_u} \cap S_{u,\prec_v}$ will be empty as well. Thus,

Observation 4.7.1 *Given that some neighbor device $y \in \prec_u$ is lost, the removal of y from $\prec_u$ won't cause any further connection closings.*

If no connection to node v exists, we know from section 4.5 that there's no need for establishing the link (u, v) as long as $v \prec_u b_{u,v}$. Thus, only links (u, v) that were not in G_{XTC} because y was the better common neighbor have to be established.

Observation 4.7.2 *given that some neighbor device $y \in \prec_u$ is lost, the removal of y from $\prec_u$ causes connection establishments to all neighbor nodes $v \in \prec_u$ with $b_{u,v} = y$.*

Therefore, we extend the formulation of the XTC algorithm by Step III) which considers device losses as shown in figure 4.16.

```
XTC Algorithm

III) Link Losses:
 4:  for (each lost link $(u, y)$) {
 5:      create connection to $y$
 6:      if (error) {
 6:          remove $y$ from $\prec_u$
 6:          $C := get\_new\_candidates(\prec_u)$
 6:          $apply\_changes(C,\ R := \emptyset)$
 7:      }
 6:      else send order to $y$
 7:  }
```

Figure 4.16: XTC: Reacting upon Device Losses

Announcing Device Losses

At first glance, reacting upon device losses seems to be very simple. But the real challenge in dealing with device losses is not about how a node u has to react when observing a device loss, it is more about assuring that all of the nodes involved get informed about the device loss. This can be seen from the example shown in figure 4.17.

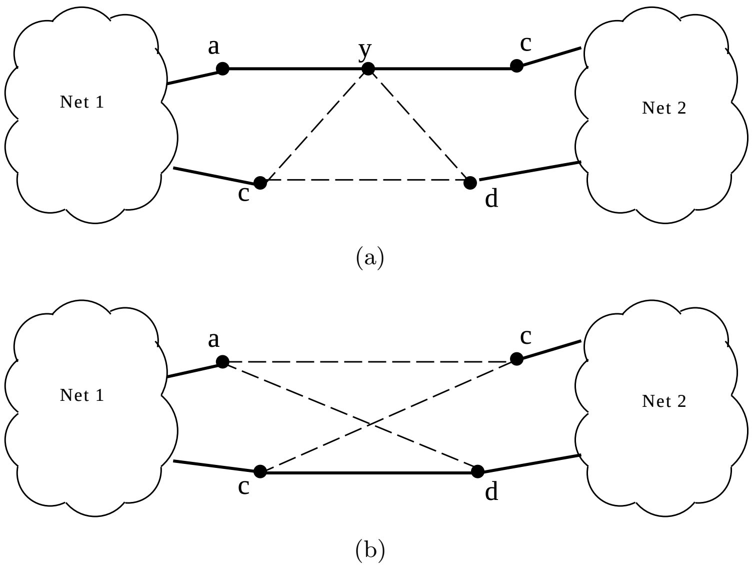


Figure 4.17: Asymmetry: Another example

Consider the network depicted in figure 4.17 a). Node y is the node that ensures that the topology control graph is connected. Note that the link (c, d) is not established because of node $b_{cd} = y$. Furthermore, assume that the links (a, c) and (c, d) are not established because better common neighbors can be found in Net 1 (Net 2 respectively).

Figure 4.17 b) shows the resulting topology control graph G_{XTC} if node u would not have been part of G_0. In this case, the link (c, d) is established because there is no common better neighbor. In contrast to figure 4.17 a), the link (a, c) is not part of the topology control graph G_{XTC} because both, node c, and node d are better common neighbors of (a, c).

Thus, by removing node y from the topology control graph shown in figure 4.17 a), we're going to receive the topology control graph depicted in figure 4.17 b) if each node applies the device loss procedure shown in figure 4.16.

But the problem is obvious: since only nodes a and c are able to observe the loss of device y directly, these nodes have to inform the remaining nodes about the device loss for that, in this case, the link (c, d) gets established by node c or node d. Thus in general, we can make the following

Observation 4.7.3 *After observing a device loss, the observing nodes have to inform the remaining nodes about the device loss observed.*

Solution

The problem of announcing a device loss is similar to the problem of bringing an updated neighbor ordering to all known neighbors, as discussed in section 4.1. But this time, we're going for the broadcast solution, because it is fast and can be implemented efficiently.

A rough draft of the solution is given in figure 4.18. The idea is quite simple: when observing a device loss (Step III, Line 4), the observing node u calls the function *remove_neighbor*() which is shown in more detail in figure 4.19: The observing node u sends a 'DEV_LOSS' packet, containing the Id of the lost device to all of its connected neighbor devices and establishes new connections if necessary (Lines 3-4 in figure 4.19).

The *remove_neighbor*() will then be executed recursively: Each receiving node of the 'DEV_LOSS' packet is going to execute the *remove_neighbor*() function as well, thus the receiving node will send the 'DEV_LOSS' packet to all its connected neighbors as well. As soon as a node u receives a 'DEV_LOSS' packet of a neighbor that cannot be found in its neighbor ordering $\prec_u$, it stops forwarding the packet.

Conclusion

1. In section 4.1 a node u has to distribute its neighbor order $\prec_u$ to all of its neighbor devices due to some changes in topology. The problem mentioned here is familiar. In fact, to ensure that the topology control graph will be connected after a device loss, all neighbors of the lost

```
XTC Algorithm

II) Select topology control neighbors:
 1:  for (each received packet) {
 2:      switch (packet type) {
  ⋮
25:      case ('DEV_LOSS' packet)
26:          if (lost device $y \in \prec_u$)  $remove\_device(\prec_u, y)$
29:      }
30:  }

III) Link Losses:
 1:  for (each lost link $(u, y)$) {
 2:      create connection to $y$
 3:      if (error) {
 4:          $remove\_device(\prec_u, y)$
 5:      else send order to $y$
 6:  }
```

Figure 4.18: XTC: Link Losses, completed

```
$remove\_neighbor(\prec_u, y)$:
 1:  remove neighbor $y$ from $\prec_u$
 2:  send 'LINK_LOSS' packet to all connected neighbors
 3:  $C := get\_new\_candidates(\prec_u)$
 4:  $apply\_changes(C,\ R := \emptyset)$
```

Figure 4.19: Reacting on device Losses

node y have to be informed about its removal from the topology graph. But since a loss device of some device y can only be observed by its connected neighbor devices, it is not guaranteed that the observing devices are able to reach *all* former neighbors of node y directly.

2. One of the reasons why we did not go for the broadcast solution in section 4.6.2 was because the broadcast solution comes along with a lot of traps in cyclic networks. But in this case, the broadcast can be implemented easily and efficiently as described in this subsection.

Chapter 5

Implementation

In this chapter an overview of the implementation of the adaptive XTC algorithm - the connection manager - is given. In section 5.1 the adaptive XTC algorithm derived during the last chapters is summarized. Section 5.2 gives an overview of the architecture chosen.

The interfaces of the connection manager are explained in detail in section 5.3, while section 5.4 essentially gives an overview of the different packet formats used. Tools and methods that were used for debugging are introduced in section 5.5.

5.1 The Algorithm

The algorithm implemented in this thesis is summarized in figures 5.1 to 5.3. Basically, this is the adaptive XTC algorithm obtained from the considerations during the previous chapter.

5.1.1 Part I: Edge Updating & Link Establishment

Part I) of the XTC algorithm (figure 5.1) will be executed forever: In Line 2, the sets R_u and C_u are initialized to be empty. In Line 3 a node u does a device discovery and writes the pair $\{id_v, \omega^*_{uv}\}$ of the devices found to the set D_u. From Line 4-6, the node examines each of the neighbor devices found, and updates the neighbor list accordingly by calling the *update_neighbor_order*() function.

This function is shown in more detail in figure 5.2: if a the neighbor node v passed cannot be found in $\prec_u$, a new entry is created. If the device was already in $\prec_u$, the neighbor list will be updated, according to the new weight ω_{uv} of the edge (u, v).

In Line 7 of Step I) the function *get_candidates*() gets called, which evaluates to which *unconnected* neighbor device a connection has to be established, given the updated neighbor order $\prec_u$. This function is described in more detail in figure 5.2, for further details refer to section 4.5. The nodes evaluated by the function get returned and stored in the set C_u (Line 7, Step I).

After evaluating new candidates, all currently connected nodes are added to the set R to cancel asymmetry, as described in section 4.6.

Afterwards, the function *apply_changes*() gets called, which establishes a connection to each new candidate (i.e. to all nodes $\in C_u$) found in Line 7, and sends the updated order to them. Additionally, it the updated order $\prec_u^*$ is sent to all connected neighbors.

Note that in comparison to the formulation presented in figure 4.12 no copy of the neighbor order before updating will be established, since this is not needed.

5.1.2 Part II: Link Selection

The second part of the adaptive XTC topology control algorithm is responsible for processing received packets, as well as to decide upon the packets received if the link (u, v) shall be kept or closed.

Lines 2-22 basically correspond to the 2-way handshake protocol explained in section 3.1.2 plus the extensions to cancel asymmetry when receiving a neighbor ordering as explained in section 4.6, that is with each neighbor ordering $\prec_v$ received, a node u reads the current link weight ω_{uv}^* and updates its neighbor ordering $\prec_u$ accordingly (Lines 4-6 in Step II).

After updating there may be new candidates / outdated candidates among all edges adjacent to node u, which are evaluated at Lines 6 and 7. Processing the candidates evaluated is done in Line 16.

From Lines 7-15, Step II) the receiving node u actually decides if the link (u, v) should be kept or not. In case of a "negative" decision (Lines 7-11), node u determines the better common neighbor $b_{u,v}$, and informs the sending node v about its decision made by sending a 'NACK' packet back to v, containing the Id of $b_{u,v}$ (refer to section 4.5 for further details). In case of a "positive" decision, the receiving node u simply sends back an 'ACK' packet (Lines 12-15).

To cancel asymmetry, a node u updates the weight ω_{uv} of a link (u, v) with each received order packet at Line 5 (refer to section 4.6 for further details).

5.1.3 Part III: Link Losses

The third part of the adaptive XTC algorithm is responsible for reacting upon link / device losses.

For each link loss observed, the observing node tries to re-establish the connection (Line 2). If reconnecting is not successful, the neighbor device is considered as malfunctioning, and thus has to be removed from the neighbor ordering (Lines 3-5).

A proper removal of some neighbor y from $\prec_u$ is obtained by calling the *remove_neighbor*() function (Line 4), which is depicted in more detail in figure 5.3: The removal of y from $\prec_u$ (Line 1), may cause edges (u, v) to become new candidates, which is determined at Line 3. A connection to each device evaluated will be established by calling the *apply_changes*() functions (Line 4).

As explained in section 4.7, device losses have to be announced because they may not be locally solvable. Therefore, a 'DEV_LOSS' packet is sent to all connected neighbors (Line 2).

XTC Algorithm

I) Broadcast $\prec_u$ to neighbor devices:

```
1:  while (true) {
2:      $R := \{\}, \ C := \{\}$
3:      discover neighbor devices: $D_u$
4:      for (each node $v \in D_u$) {
5:          $update\_neighbor\_order(\prec_u, v)$
6:      }
7:      $C := get\_candidates(\prec_u)$
8:      $R :=$ all connected neighbors in $\prec_u$
9:      $apply\_changes(\prec_u, R, C)$
10: }
```

II) Select topology control neighbors:

```
1:  for (each received packet) {
2:      switch (packet type) {
3:      case (order packet $\prec_v$)
4:          $R := \{\}, \ C := \{\}$
5:          $update\_neighbor\_order(\prec_u, v)$
6:          $C := get\_candidates(\prec_u)$
7:          $R := get\_outdated\_candidates()$
8:          if ($S_{v,\prec_u} \cap S_{u,\prec_v} \neq \emptyset$) {
9:              choose better common node $b_{uv} \in (S_{v,\prec_u} \cap S_{u,\prec_v})$
10:             store better common node $b_{uv}$ in $\prec_u$
11:             send 'NACK' packet to $v$
12:         }
13:         else {
14:             send 'ACK' packet to $v$
15:             inform application layer
16:         }
17:         $apply\_changes(\prec_u, R, C)$
18:     case ('NACK' packet)
19:         store better common node $b_{u,v}$
20:         disconnect
21:     case ('ACK' packet)
22:         inform application layer
23:     case ('DEV_LOSS' packet)
24:         if (lost device $y \in \prec_u$) $remove\_device(\prec_u, y)$
25:     }
26: }
```

III) Link Losses:

```
1: for (each lost link $(u, y)$) {
2:     create connection to $y$
3:     if (error) {
4:         $remove\_device(\prec_u, y)$
5:     else send order to $y$
6: }
```

Figure 5.1: Implemented XTC algorithm

XTC Algorithm: Evaluation Functions

```
update_neighbor_order($\prec_u, v$):
1: if ($v \notin \prec_u$) {
2:     $\prec_u = \prec_u \cup v\,(id_v, \omega_{uv}, b_{uv} := \emptyset)$
3: }
4: else {
5:     update $\prec_u$ according to $w^*(u,v)$
6: }
7: return $R$

get_candidates($\prec_u$):
1: $C := \{\}$
2: for (each unconnected node $v \in \prec_u$) {
3:     if (($v \prec_u b_{u,v}$) | ($b_{u,v} = \emptyset$))   $C := C \cup v$
4: }
5: return $C$

get_outdated_candidates():
   refer to appendix ??
```

Figure 5.2: Implemented XTC algorithm, Evaluation Functions

XTC Algorithm: Apply Functions

```
apply_changes($\prec_u, R, C$):
1: for (each $v \in C$) {
2:     establish connection to $v$
3:     send $\prec_u$ to $v$
4: }
5: for (each $v \in R$) {
6:     send $\prec_u$ to $v$
7: }

remove_neighbor($\prec_u, y$):
1: remove neighbor $y$ from $\prec_u$
2: send 'DEV_LOSS' packet to all connected neighbors
3: $C$ := get_new_candidates($\prec_u$)
4: apply_changes($C$, $R := \emptyset$)
```

Figure 5.3: Implemented XTC algorithm, Apply Functions

5.2 Architecture Overview

Figure 5.4 shows the main components the implementation consists of. Basically, the components can be divided into three parts:

1. *Functional Parts (Threads)*: The functional parts execute the XTC algorithm as shown in figure 5.1. They are represented by “clouds”.
2. *Global Data Structures*: The global data structures of the implementation are labeled as ’neighbor_order’, ’send_buf’, ’con_buf’, and ’inqres’.
3. *Data Access Functions*: Usually threads do not access global data structures directly - they make use of special data access functions. In figure 5.4 these functions are represented by a small box labeled with the name of the function. A thread that makes use of a specific data access function is indicated by a dashed arrow.

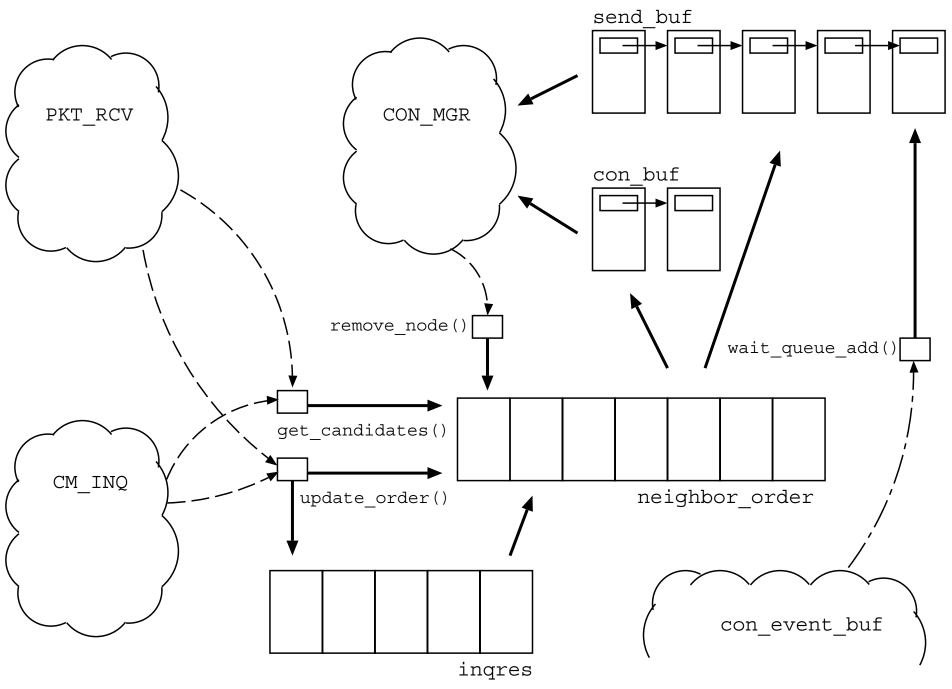


Figure 5.4: Implementation: Architecture

5.2.1 Functional Parts

The functionality of the XTC algorithm was divided into four smaller functional parts, as shown in figure 5.4.

'CM_INQ' Thread The 'CM_INQ' thread (inquiry thread) is permanently executing Step I) of the XTC algorithm presented in figure 5.1. For updating the neighbor order and getting the new candidates the inquiry thread calls the data access functions *update_order*() and *get_candidates*().

'CM_PKT_RCV' Thread The 'CM_PKT_RCV' (packet receiving) thread is permanently executing Step II) of the XTC algorithm in figure 5.1. Thus, the packet receiving thread is most of the time just waiting for incoming packets, which then are processed immediately corresponding to Step II) of the XTC algorithm (figure 5.1.

'CON_MGR' Basically, the 'CON_MGR' (connection management) thread is the implementation of the *apply_changes*() function shown in figure 5.3. The nodes of the set C (see figure 5.3) are contained in the 'con_buf' data structure, the nodes of the set R in the 'send_buf' data structure (see next subsection). Thus, the connection management thread simply tries to close / establish new links corresponding to the nodes contained in the buffers 'send_buf', and 'con_buf'.

If a link (u, v) cannot be established, the neighbor v has to be removed from $\prec_u$ (see below). This is done by calling the function *remove_neighbor*(), which corresponds to the *remove_neighbor*() function shown in figure 5.3.

'con_evt_buf' The 'con_evt_buf' (connection event buffer) thread gets active each time a connection change occurs (e.g. if a new connection was established by some remote device, or if a connection to a neighbor device was lost). Essentially, this thread informs the application layer when connections get closed (refer to section 5.3), and executes Step III) of the XTC algorithm presented figure 5.1. Instead of executing Step III) directly, the connection event buffer thread simply adds the node v of the lost link (u, v) to the 'con_buf' by making use of the function *wait_queue_add*().

5.2.2 Data Structures

The implementation of the XTC algorithm consists of the following global data structures:

neighbor_order The neighbor order $\prec_u$ of a node u. Basically, the neighbor order structure is just a sorted list who's elements are *neighbors* (see below).

neighbor A neighbor node v that is in $\prec_u$. Basically, a *neighbor* element consists of the Id of the neighbor node v, the weight ω_{uv} of the link (u, v), and the better common neighbor $b_{u,v}$ (if there is any).

inqres An array to which the discovered neighbor devices found during neighbor discovery are written to. Hence, the *inqres* array corresponds to the set D_u in the algorithm shown in figure 5.1.

con_buf The set of candidates C. A linked list FIFO queue, containing *neighbor* elements was chosen to implement the set C.

send_buf The set of outdated candidates R. A linked list FIFO queue, containing *neighbor* elements was chosen to implement the set R.

5.2.3 Data Access Functions

The most important data access functions the functional parts make use of are the following:

***update_order*($\prec_u, id_v, \omega_{uv}$)** The *update_order*() function updates the order $\prec_u$, given the Id of a node v and the updated weight ω_{uv} of the link (u, v) as shown in figure 5.2. Instead of collecting the outdated candidates R and returning them to the calling thread, the outdated candidates evaluated are added directly to the 'send_buf'.

***get_candidates*($\prec_u$)** The *get_candidates*() function evaluates new candidates by inspecting the neighbor order $\prec_u$ passed as described in figure 5.2. Instead of collecting the new candidates C and returning them to the calling thread, the new candidates evaluated are added directly to the 'con_buf'.

***remove_neighbor*($\prec_u, id_y$)** The *remove_node*() function removes node y from the neighbor list $\prec_u$, as shown in figure 5.2. It is exclusively called by the connection management ('CON_MGR') thread.

***wait_queue_add*(&wait_buf, v)** The *wait_queue_add*() function adds a *neighbor* element to the FIFO queue specified by *&wait_queue*, that is either to the 'send_queue' or the 'con_queue'.

***wait_queue_remove*(&wait_buf, v)** The *wait_queue_remove* function removes an element from the queue specified. Usually, this function is called by the connection management thread if a node v has been processed successfully (i.e. a connection to v has been established, or the neighbor order has been sent to v successfully.

5.2.4 Thread Timing

This section shall give a brief overview of when, and how often the threads shown in figure 5.4 are executed.

Inquiry Thread

The inquiry thread periodically executes Step I) of the XTC algorithm shown in figure 5.1. As discussed in section 3.4, it makes sense to decrease the frequency of executing the while loop after a few cycles for that excessive power consumption can be reduced. On the other hand, a node should discover all of its neighbor devices as soon as possible right after power-up for that it can be integrated as fast as possible into the topology control graph.

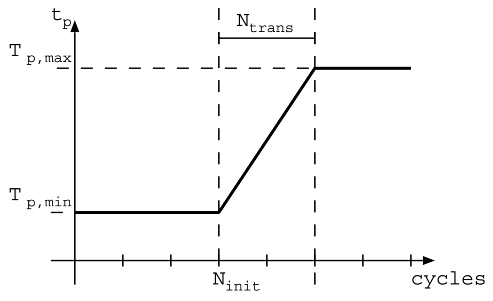


Figure 5.5: Controlling the period of the inquiry thread

Therefore, the period of the inquiry thread can be controlled as shown in figure 5.5: After power-up, the update period will be set to $T_{p,min}$. After executing the while loop in Step I) of the XTC algorithm N_{init} times, the period of executing Step I) will be increased after each cycle, until the maximal period $T_{p,max}$ has been reached. To control the slope of the period, the number of transition steps N_{trans} can be controlled as well.

Since these parameters directly affect the behavior of the network during establishment, these parameters have to be available right after power-up. For that the user is able to control these parameters they can be written directly into the EEPROM of the BTnode from where they are read right after power-up. Additionally, they can be adjusted at runtime using the terminal (see section 5.3).

Packet Receiving Thread

The packet receiving thread is signaled by the BT Stack each time a packet from a connected neighbor device arrives. Thus the packet receiving thread gets woken up randomly.

Connection Event Buffer

As the packet receiving thread, the connection event buffer is signaled randomly with each occurring connection event.

'Connection Management Thread

The connection management thread is woken up each time a change in the neighbor ordering occurred, that is if one of the following events occur:

1. After the inquiry thread has updated the neighbor order
2. After a packet was received from a neighbor device
3. If a link loss was observed.

Since all of these events are observed from other threads, these threads have to wake up the connection management thread as described in the next section. Also, active time points of the 'CON_MGR' thread are not deterministic.

5.2.5 Thread Interaction

Basically, the implementation makes use of three different mechanisms for controlling interaction of the four threads presented in figure 5.4. These are:

'con_mgr_start' Event Each time a new connection has to be established (i.e. each time a new candidate has been determined), the connection management thread has to be woken up for that it starts to empty the 'con_buf' and the 'send_buf'. This can be done by posting the 'con_mgr_start' event. As can be seen from the XTC algorithm (figures 5.1 to 5.3), all of the remaining threads have to call the *apply_changes*() function, thus all of the remaining threads may post the 'con_mgr_start' event.

'data_mutex' Since all threads have to access the data structures shown in figure 5.4, and since all threads (except the inquiry thread) do more or less have to access that data at random times, the data structures are kept consistent by making use of mutual exclusion.

'bt_module_mutex' The BTnode is not able to establish a connection to another device during neighbor discovery. But the packet receiving thread, as well as the connection event buffer wake up the 'CON_MGR' thread each time a significant change in the local order was observed. For that the inquiry thread cannot be interrupted and vice versa, another mutex is used that grants access to only one of the two threads.

5.2.6 Thread Contexts

The reason for the introduction of the connection management thread is as follows:

A connection establishment may take a long time[1]. Hence, a thread calling the *apply_changes*() will be blocked for a very long time, depending on the number of connection establishments that have to be done (i.e. depending of the number of nodes contained in the 'con_queue'. Hence, if e.g. the packet receiving thread would call the *apply_changes*() function, it would not be able to process incoming packets during connection establishment. Since these packets are buffered in the lower level BT Stack (refer to section 5.3, a buffer overflow could not be avoided[2].

It's clear that by introducing the connection management thread together with the send queue and the connection queue, buffer overflows can be avoided.

5.3 Interfaces

An overview of the interfaces our implementation of the XTC algorithm makes use of is given in figure 5.6. As can be seen from figure 5.6, a layered approach has been chosen for providing an interface for applications.

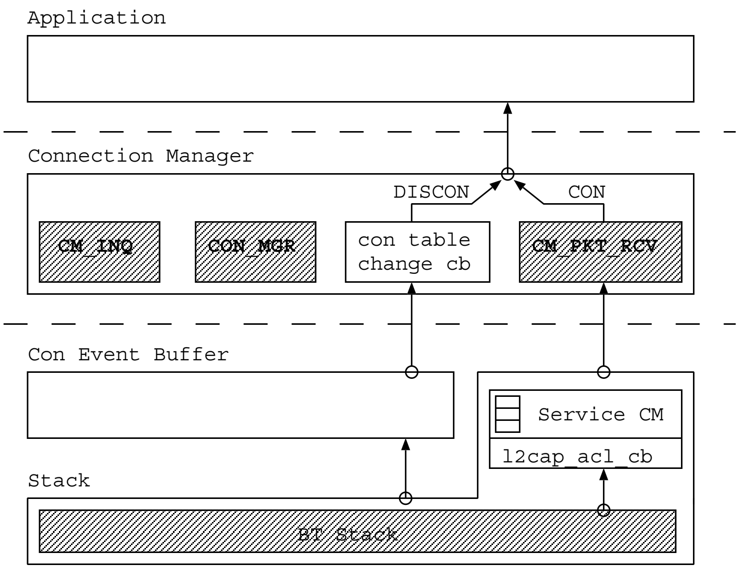


Figure 5.6: Connection Manager: Interfaces

Hence, an application, making use of the XTC implementation provided in this thesis consists of the following four components:

Application The application itself.

[1]up to several seconds, as we shall see in section 6.2

[2]the same problem occurs with connection events

Connection Manager The implementation of the adaptive XTC algorithm presented in figures 5.1 to 5.3. The connection manager informs the application each time a *reliable* connection has been established, i.e. the application is not informed before the link has been acknowledged by both parties. Likewise, the application is informed only if a *reliable* connection was closed (i.e a previously acknowledged connection was closed).

BT Stack The Bluetooth stack. It calls the higher layers if a connection event (e.g. a new connection has been established / closed) occurs, or if a packet from another device has been received.

Con Event Buffer Receives connection events from the BT Stack, stores them in a FIFO queue and signals an Event to the higher Layer. This enables a short execution time of the callback function running with the high priority of the BT Stack thread.

5.3.1 Application Interface

This section shall give an overview of the application interface provided by the connection manager.

Initialization

To make use of the connection manager, the application has to initialize the connection manager by calling the *con_mgr_init*() function. This function essentially initializes the connection manager stack, i.e. the global variables needed by the connection manager.

Callback Registration

For that the connection manager is able to inform the application about connection events, a callback function has to be registered that is called by the connection manager each time a *reliable* connection has been established or closed. This can be done using the *con_mgr_register_rel_con_change_cb*() function.

A more detailed view of when the connection event callback is called is given in figure 5.7: in case of a new connection establishment, the application is signaled not before either an 'ACK' packet has been received, or an 'ACK' packet has been sent successfully. Hence, it's the packet sending thread that announces reliable connections (refer to section 5.2.1, figure 5.1, and figure 5.4).

The closing of a reliable connection is announced right after the BT Stack (the Con Event Buffer resp.) signaled a disconnection event (refer to figure 5.6. The connection manager signals the application layer only if the disconnection event triggered by the Con Event Buffer affects a reliable connection.

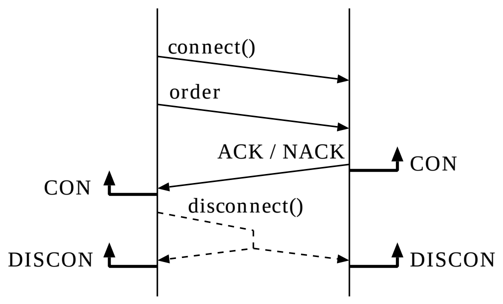


Figure 5.7: Calling the *rel_con_change_cb*() callback

Parameter Settings

An application may control the connection manager at runtime by setting various parameters. An overview of the parameters and their effects is given in table 5.1.

Connection Info Packet

An application may be interested in the neighbor order established by the connection manager. For that the neighbor order can be obtained also from remote devices, we decided to make this information available in form of a packet called the *Connection Info Packet* (CIP). This packet may be obtained by the application by calling the *con_mgr_fill_con_info*() function.

Entries	BT Addr$_1$	State$_1$
RSSI$_1$	BT Addr$_2$	State$_2$
RSSI$_2$	...	

Figure 5.8: Format of an order packet

The packet format is shown in figure 5.8: The first byte corresponds to the number of neighbors that actually can be found in the neighbor order $\prec_u$ of some node u. Then, an entry for each neighbor device v is made, containing the following information:

1. *BT Addr* (6 Bytes) The BT address of the neighbor device v,

2. *State* (1 Bytes) The current connection state. Possible values may be:

   0: Unconnected

Param	**Function**	**Description**
$T_{p,max}$	cm_set_ai_min_period()	Sets the minimal period of the inquiry thread as described in section 5.2.4, figure 5.5.
$T_{p,max}$	cm_set_ai_max_period()	Sets the maximal period of the inquiry thread as described in section 5.2.4, figure 5.5.
N_{init}	cm_set_ai_init()	Sets the number of initial inquiries done by the inquiry thread as described in section 5.2.4, figure 5.5.
N_{trans}	cm_set_ai_trans()	Sets the number of inquiries done by the inquiry thread during transition phase as described in section 5.2.4, figure 5.5.
$\sigma_{T_{inq}}$	cm_set_ai_max_dev()	Maximal deviation of the inquiry period. Used to prevent inquiry collisions (refer to section 6.1 for more details).
T_{inq}	cm_set_ai_time()	Sets the duration of an inquiry (i.e. neighbor discovery process).
ai	cm_set_autinq()	Turns on / off automatic inquiry.
$\Delta RSSI_{id}$	cm_set_rssi_id()	Defines the minimal RSSI difference needed for that the weights of two different links are not considered as equal (refer to section 6.4) for further details).
$\Delta RSSI_{th}$	cm_set_rssi_update()	Defines the minimal RSSI difference needed for that the weight of a link ω_{uv} is updated (refer to section 6.4 for further details).
N_{con}	cm_set_max_cons()	Defines the maximal number of connection tries done by the 'CON_MGR' thread before considering the neighbor as not responsive.
$\sigma_{T_{con}}$	cm_set_con_max_per()	Maximal wait time before a new connection is established. Used to reduce congestion (refer to section 6.3, section 6.4 respectively).
T_{cpen}	cm_set_con_coll_pen()	Wait time after a page collision occured (refer to section 6.2.2).
T_v	cm_set_vis_time()	Wait time between an inquiring and paging (refer to section 6.4)
α	cm_set_lpf_alpha()	Low pass filter coefficient α (refer to section 6.5)
β	cm_set_lpf_beta()	Low pass filter coefficient β (refer to section 6.5)

Table 5.1: Connection Manager Paramters

   1: Connected, my role MASTER

   2: Connected, my role SLAVE

3. *RSSI* (1 Bytes) The weight of the edge (u, v), measured by node u (i.e. the RSSI value measured by node u).

5.3.2 User Interaction

For user interaction, the BTnut OS provides a thread called the *terminal thread*. Basically, this thread is just an implementation of a simple command line interface to the BTnode, which allows the user to interact with the BTnode via a terminal.

The connection manager provides several terminal commands for:

- setting the parameters introduced in section 6.5.2
- writing the parameters introduced in section 6.5.2 to the EEPROM
- easy debugging

To make use of the terminal commands provided by the connection manager, an application has to:

1. initialize the terminal thread (refer to the online API of the Nut OS [8]).
2. initialize the connection manager, as explained in section 5.3.1.
3. register the commands at the terminal by calling the *con_mgr_register_cmds*() function.

A summary of the commands provided by the connection manager is given in appendix A.

5.3.3 Organization

To improve maintainability, the source code was divided into four different files. An overview is given in the table below:

File	**Description**
con_mgr_xtc.c con_mgr_xtc.h	Implementation of the connection manager.
srtd_list.c srtd_list.h	Implementation of an indexed sorted list, using pointers. It was tried to implement the sorted list in a somewhat “object-oriented” fashion, thus, the implementation of the sorted list provided should not only be usable for the implementation of the neighbor order - it should be usable for other applications as well.
wait_queue.c wait_queue.h	Implementation of a FIFO wait queue / Stack in form of a linked list. As well, this implementation should be usable for other applications as well.
cm_cmds.c	Implements the terminal commands for controlling / debugging the connection manager. Refer to section 5.3.2 and 5.5 for further details.

5.4 Communication

As mentioned before, a node communicates with its neighbor as defined by the 2-way handshake protocol introduced in section 3.1.2. In this section we’re going to give an overview of the different packet formats used.

5.4.1 Order packet

After establishing a connection to a neighbor node v, the initiating node u sends its order $\prec_u$ to the neighbor node v. The format of an order packet is shown in figure 5.9, a description of the fields it consists of can be found in table 5.2.

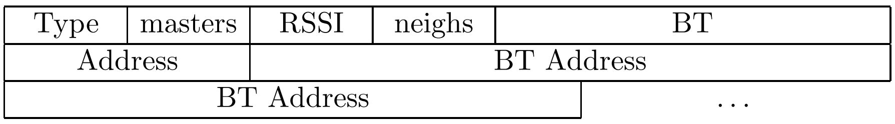


Figure 5.9: Format of an order packet

5.4.2 ’ACK’ Packet

Sent as a response to a received order packet. Indicates that the link over which the packet was received has to be in G_{XTC} and thus has to be kept.

Field	Size	Description
Type	1 Byte	Type of this packet. Set to 'ORDER'.
Masters	1 Byte	Number of master devices. Needed for ressource sharing (refer to section 6.7).
RSSI	1 Byte	The RSSI value measured by the sender of the order packet (corresponds to the weight ω_{uv}). Needed for cancelling asymmetry as described in section 6.6.
Neighs	1 Byte	Number of neighbor devices the order consists of.
BT Address	Neighs $\cdot$ 6 Bytes	Id's of the neighbor devices of the neighbor order. A bluetooth Id has a length of 6 bytes.

Table 5.2: Fields of an order packet

The 'ACK' packet only consists of the 'Type' field, which is set to 'ACK'.

5.4.3 'NACK' Packet

A 'NACK' packet is sent as a response to a received order packet. Indicates that the link over which the packet was received should be closed. The format of an 'NACK' packet is shown in figure 5.10, a description of the fields it consists of is given in table 5.3.

Type	Reason	Better Common Neighbor

Figure 5.10: Format of a 'NACK' packet

5.4.4 'DEV_LOSS' Packet

A 'DEV_LOSS' packet is sent after a device loss was observed. A 'DEV_LOSS' consists of a type field and the BT address of the lost node, as shown in figure 5.11

Type	Lost Device

Figure 5.11: Format of a 'DEV_LOSS' packet

Field	**Size**	**Description**
Type	1 Byte	Type of this packet. Set to 'NACK'.
Reason	1 Byte	The reason why the connection should be closed. Usually, the reason is that the link should not be in G_{XTC}. A summary of further reasons is given in table 5.4 below.
Better Common Neighbor	6 Bytes	If the reason for connection closing is set to 'CON_RQST_NOT_ACCEPTED', the BT address of the better common neighbor can be found in this field.

Table 5.3: Fields of a 'NACK packet

Reason	Description
CM_CON_NOT_ACCEPTED	Connection not accepted due to XTC link selection criteria
CM_RESOURCES_EXHAUSTED	Indicates that connection should be closed because there are not enough resources available to keep it.
CM_ROLE_SWITCH_FAILED	Indicates that connection cannot be kept because a necessary role switch failed

Table 5.4: Reasons for not acknowledging a connection

5.5 Debugging

Basically, debugging was done using three different techniques: by making use of the terminal, by assigning LED patterns to the most important operations the Bluetooth device, and by making use of the JAWS GUI.

5.5.1 The Terminal

Error messages and operating states are printed to the terminal (refer to section 5.3.2). Additionally, it is possible to execute the XTC algorithm step by step by making use of several terminal commands. An overview can be found in appendix A.

5.5.2 LED Patterns

For that operation states of the nodes can be observed without making use of the terminal (for which a node has to be attached to a computer), they are represented by different LED Patterns. An overview is given in the following table:

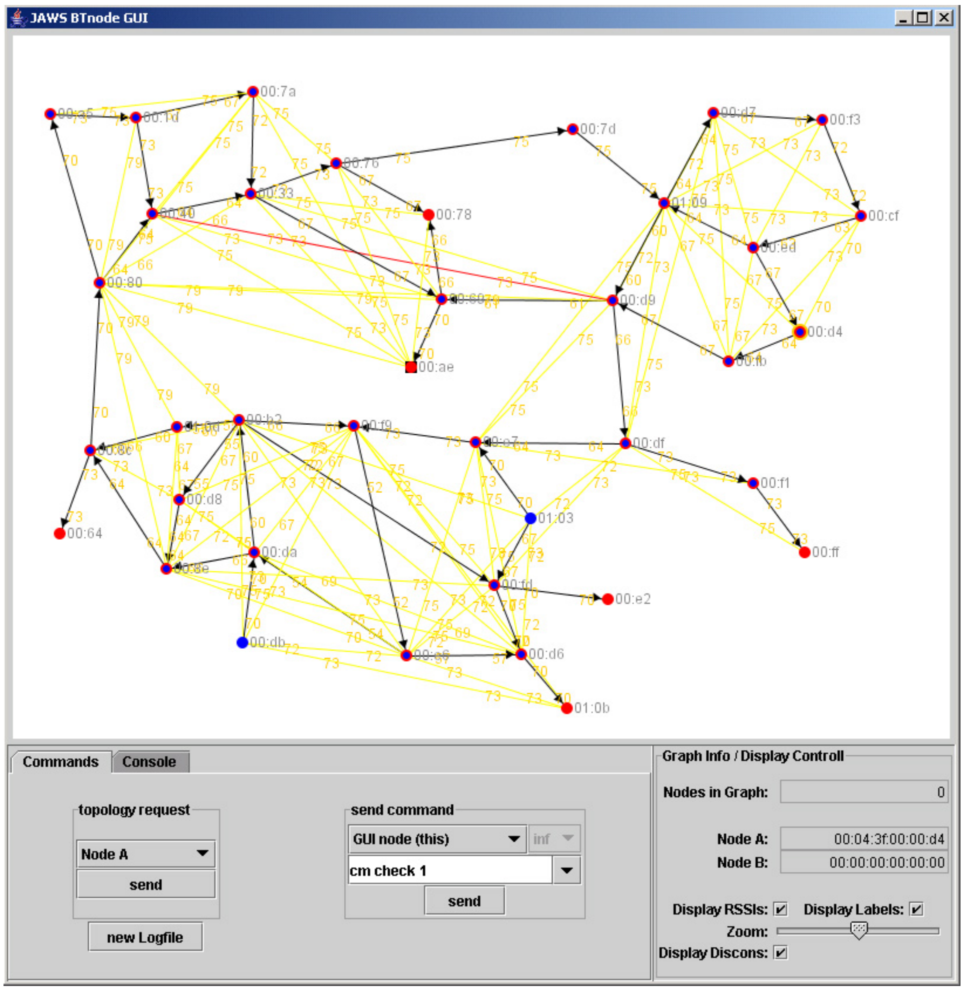
Pattern	Description
green	Inquiring
yellow	Connecting
red	Role Switching
green & yellow	Packet receiving thread is active
blue & red	Sending order packet to a neighbor device
"Knight Rider"	Reliable connection has been established

5.5.3 Jaws-GUI

The most important tool used for debugging was the JAWS GUI that comes together with the JAWS application [6].

Essentially, the JAWS application automatically forms a connected network of Bluetooth devices (using the connection manager), and provides services for transparent ad-hoc connections from a host device to a number of target devices. A graphical user interface to these services is given by the JAWS GUI, which additionally provides a graphical representation of the network. A snapshot of the JAWS GUI is given in figure 5.12.

To establish the graphical representation of the network established by the connection manager, the JAWS GUI simply collects the *connection info packets* of all nodes in the network.



GUI node (this)

inf

cm check 1

Nodes in Graph: 0

Node A: 00:04:3f:00:00:d4

Node B: 00:00:00:00:00:00

Display RSSIs: ✓ Display Labels: ✓

Zoom:

Display Discons: ✓

Figure 5.12: JAWS GUI: Snapshot

Chapter 6

Implementation Issues

This chapter gives an overview of several measures that had to be taken for that the mapping of the adaptive XTC algorithm onto the BTnode platform became possible. Hence, this chapter covers the problems occurred during implementation of the algorithm and the countermeasures taken, which shall point out that implementation complexity is bigger than maybe expected after reading the few lines of pseudo-code of the initial XTC algorithm introduced in section 2.2, figure 2.1. Furthermore, the missing details of the implementation overview presented in the previous chapter are explained here.

Section 6.1 covers the problem of the inquiry operation in Bluetooth devices, while section 6.2 deals with the connection establishment procedure in Bluetooth. In section 6.3 it is shown that the performance of network establishment is poor if several devices are trying to establish a connection at the same time. Finally, section 6.4 discusses network establishment in more detail.

The measures taken to reduce sensitivity of the adaptive XTC algorithm are explained during section 6.5. Afterwards, the countermeasures taken due to asymmetric link weight measurements are explained.

In section 6.7 it is explained how the adaptive XTC algorithm tries to bypass the restrictions given in Bluetooth scatternets, while in section 6.8 the consequences of a bounded neighbor order are discussed.

6.1 Inquiring

The Bluetooth devices used in this thesis come along with the following restriction: if a device is inquiring (i.e. doing neighbor discovery), the device is not able to respond to inquiries done by other devices. In other words: if two BTnodes are inquiring at the same time, they won't discover each other.

The resulting problem for our connection manager is obvious: if each device is executing the adaptive XTC topology control algorithm depicted in figure 5.1 using the inquiry thread timing shown in figure 5.2.4, and if we would power up the nodes of an unconnected network exactly at the same time, none of the nodes would discover its neighbor nodes at any time.

To solve this problem, we introduced the parameter $\sigma_{T_{inq}}$, which is the maximal value a random variable t_{dev} can take. More precisely, t_{dev} is a uniformly distributed random variable in the interval $[-\sigma_{T_{inq}}, +\sigma_{T_{inq}}$ with mean zero. By adding t_{dev} to the current period T_p of the inquiry thread, inquiry phases of neighbor devices get distributed randomly in time. This is shown in the digram in figure 6.1.

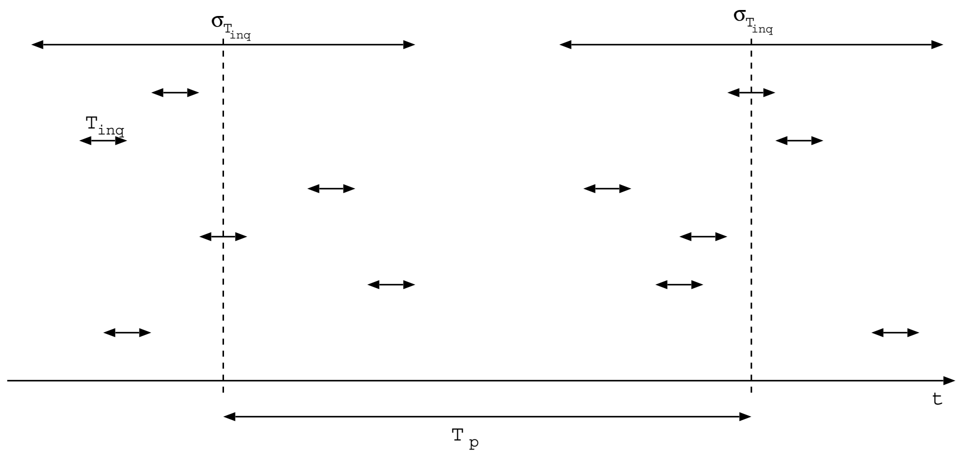


Figure 6.1: Random inquiry timing

It's clear that by increasing $\sigma_{T_{inq}}$ the probability of detecting "most of" neighbor devices increases, but also the setting of the inquiry duration T_{inq} plays an important role. Thus, the settings of the parameters T_p, T_{inq}, and $\sigma_{T_{inq}}$ essentially determine how fast the nodes of an unconnected network are going to discover their surroundings.

6.2 Paging

The process of establishing a connection to another device is a quite complex process in Bluetooth, and is called *paging*. For that a Bluetooth device is not paging infinitely if the desired communication partner is not responsive, the paging process will be stopped after some predefined amount of time t_{page}[1]. Thus, in Bluetooth, if a device v is not able to respond to a paging device u, a *page timeout* occurs at device u.

6.2.1 Inquiring vs. Paging

The first problem of paging in Bluetooth is the following: if a device is inquiring, it is not able to respond to a paging device. Thus, in case of a page timeout, the paging device is not able to distinguish if the desired communication partner is actually malfunctioning and not able to respond, or if the remote device was just doing an inquiry.

From an algorithmic point of view, it is clear that a malfunctioning device has to be interpreted as a device loss, and needs to be removed from the neighbor order for that the network is connected (as explained in section 4.7). But if a node is not able to distinguish between a malfunctioning device and an inquiring device, a lot of actually perfectly functioning devices would be removed from the neighbor order as well.

Of course, the connection to a previously removed, well-functioning device will be established at some late date since the inquiry thread is going to (re)discover the device repeatedly. But the removal, as well as the addition of a device to (from) the neighbor order produces a lot of unnecessary overhead. Thus, something has to be done to preserve the removal of functioning devices from the neighbor order in case of a page timeout, i.e. something has to be done to distinguish between an inquiring device and a malfunctioning device.

Basically, there are two different approaches:

Retries Instead of removing the neighbor device directly from the neighbor order after a page timeout occurred, a device tries to establish the connection again some moments later.

Page Timeout Increase By choosing $t_{page} \gg T_{inq}$, a connection can be established even if the desired communication partner is inquiring.

Note that both approaches have an effect on the reaction time of the network in case of a device loss, since a link loss discovering device tries to re-establish the broken link before considering the device as malfunctioning. A more detailed discussion of the two possibilities presented will be done in section 6.4.

[1]refer to [2] for further details

6.2.2 Paging vs. Paging

Another problem of paging is the following: a paging device is not able to respond to paging devices. Thus, if two devices are paging to each other at the same time, both devices end up with a page timeout. Thus, it is clear that in this case re-paging is not useful - the two devices are going to collide again.

Hence, the only way to solve this problem is the following: one of the two devices has to wait, and let the other device establish the connection. Fortunately, this can be done easily, since the devices know the Id of each other. Thus, the device with the lower Id waits for some predefined amount of time T_{cpen}, while the device with the higher Id pages the device with the lower Id immediately again.

6.3 Congestion

If we take a look at the adaptive XTC algorithm shown in figure 5.1, and take into account that paging devices are not able to answer to paging devices, another problem is obvious: since each device has to establish a lot of connections, it is not avoidable that a paging device tries to page a device that is either paging as well or doing an inquiry. Hence, after a while, all devices are paging at the same time thus none of them will be able to establish a connection.

An Example

An example is given in figure 6.2. Consider e.g. the third device, which is the first device that has done its inquiry. Hence, it starts to establish connections to each new candidate. At first, the third device establishes a connection to device 1, which is no problem since device 1 is idle. Next, device 3 tries pages device 2, which is not able to respond, because it is doing an inquiry. Here, we've chosen $t_{page} \approx T_{inq}$, thus a page timeout occurs.

Afterwards, device 3 pages device 5, which is doing an inquiry. Although there would be enough time for device 5 to respond to device 3 after the inquiry is done, a page timeout will occur, because device 5 immediately starts paging to device 1 after the inquiry. Note that after a while, most of the devices are paging, thus none will be able to establish a connection anymore.

Minimal Visibility Time

To solve this problem, another parameter was introduced: the *minimal visibility time T_v*. This parameter defines how long a device has to be visible for that other devices are able to establish a connection to it. Hence, the

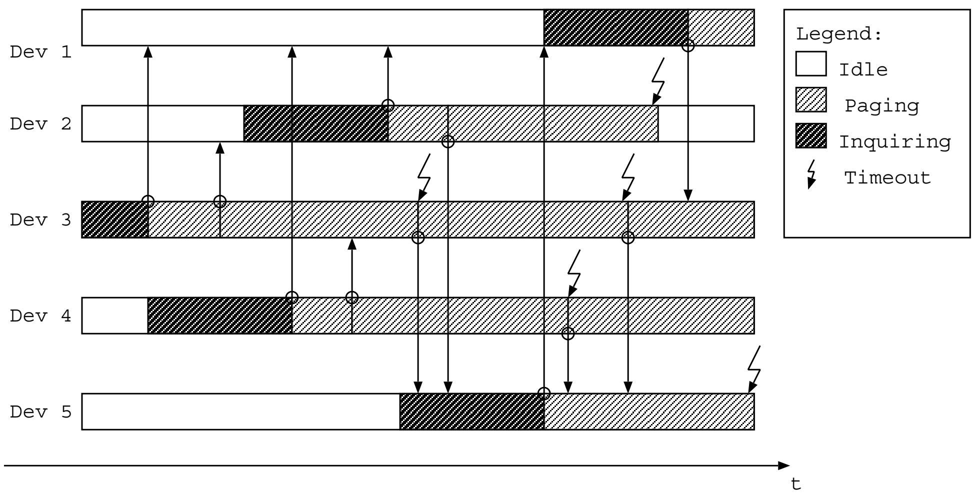


Figure 6.2: Congestion and resulting page timeouts: an example

device has to sit idle for that amount of time, that is it will neither page another device, nor doing an inquiry during that amount of time.

A device is made visible each time the device has paged another device or done an inquiry. Thus, if a page timeout occurs at some device v, neighbor devices paging v should be able to establish the connection because v sits idle for T_v after the page timeout occurred. This will at least avoid that at some time all devices are paging and thus none of them would be able to establish a connection.

Avoiding Congestion

To avoid, or at least to reduce congestion, a *relaxed timing scheme* has to be applied, that is congestion can only be reduced significant if paging collisions are reduced.

Therefore, the parameter $\sigma_{T_{con}}$ has been introduced, which is familiar with the parameter $\sigma_{T_{inq}}$ introduced in section 6.2: instead of just being visible for T_v between two connection establishments, each device extends the visibility time by some random amount of time t_{rand}. The maximal value t_{rand} can take is equal to $\sigma_{T_{con}}$.

Thus, in the same way as we've tried to avoid inquiry collisions between neighbor devices, we try to avoid page collisions between neighbor devices, and thus reduce congestion, by distributing paging phases of neighbor devices randomly in time.

6.4 Network Establishment

From the discussion during the last section, it's clear that network establishment in XTC is a quite complex process with a lot of different parameters that have to be chosen carefully for that network establishment becomes possible at all.

Basically, for that network establishment is proper and fast, a device has to establish a connection to each of its neighbor devices as soon as possible. Thus, from point of view of some device v being idle is not efficient. But from an outstanding point of view, v should be visible for other devices for "most of the time" for that congestion can be reduced / avoided. Hence, there's a trade-off in choosing the parameters $\sigma_{T_{con}}$, $\sigma_{T_{inq}}$, and T_v.
To reduce congestion, we have to:

1. Increase $\sigma_{T_{con}}$,
2. decrease the page timeout,
3. decrease the number of connection establishment retries,
4. avoid the removal of well-functioning devices from $\prec_u$.

But to avoid the removal of well-functioning devices from $\prec_u$, we have to:

1. Increment the number of retries,
2. increment the page timeout

Therefore, another parameter N_{con} has been introduced which defines the number of page trials that are done before a device is considered as malfunctioning and thus is removed from the neighbor order. Note that we did not introduce another parameter for the page timeout since it can be chosen by making use of the HCI functions provided by the BT Stack [8].

The approach chosen in this thesis to obtain a fast network establishment phase was to preserve the removal of well-functioning devices from $\prec_u$ efficiently. Thus, we've decided to choose the page timeout $t_{page} \gg T_{inq}$, and set the number of page trials N_{con} to 2.

Setting N_{con} to 2 only reduces the removal of well-functioning devices from $\prec_u$ if "direct" paging collisions are avoided, that is if the probability of two devices paging each other at the same time is minimal. This can be reached by applying a *relaxed timing scheme*, that is by choosing $\sigma_{T_{con}}$ generously. To solve the resulting conflict of "direct" paging collisions, T_{cpen} was set to $\approx t_{page}$.

6.4.1 Relaxed Timing Scheme

The timing scheme chosen in this thesis can be summarized as shown in figure 6.3: The first time line corresponds to the timing of the inquiry thread, while the second time line corresponds to the timing of the connection management thread. Idle times of the device are drawn by a dashed line, while the times the device is busy are drawn bold.

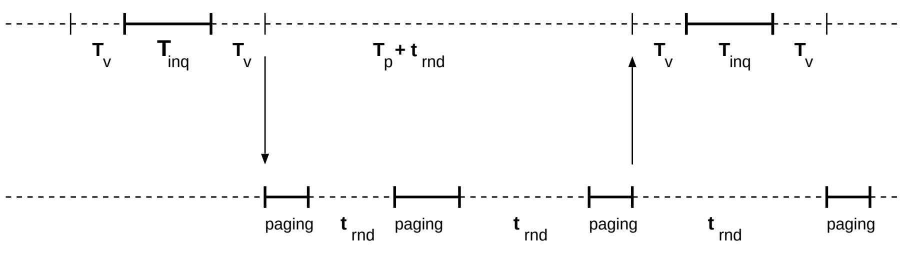


Figure 6.3: Connection Manger Timing Scheme

Basically, two consecutive inquiries are separated in time by the current value of T_p which can be chosen by the user / the application as explained in section 5.2.4, figure 5.5. To preserve inquiry collisions, a random amount of time is added to T_p who's maximal value is defined by $\sigma_{T_{inq}}$. To guarantee visibility of the device after / before an inquiry, the device waits for T_v, thus the device is at least visible for $T_p + T_v + t_{rnd}$ between two inquiries.

To reduce congestion, connection establishments to two different devices are separated by some random amount of time t_{rnd} with maximal value $\sigma_{T_{con}}$. Note that basically, the inquiry thread "interrupts" the connection management thread, that is although some connection establishments have to be done, an inquiry will be done if the inquiry thread is "ready" (that is $T_p + T_v + t_{rnd}$ has been exhausted).

Since an inquiring device is not able to page another device at the same time (and vice versa), an inquiry can be delayed because the device is still paging. This is indicated by the arrows in figure 6.3, and can be reached by making use of the 'bt_module_mutex' explained in section 5.2.5.

6.5 RSSI Fluctations

During first experiments with the first implementations, it turned out that a static network did not settle down, i.e. the topology control graph actually changed almost after every inquiry done by some node.

The main reason for this behavior is twofold:

1. the quality measure of a link provided by the Bluetooth device on the BTnode is extremely fluctuating, as shown in more detail in section 7.2, chapter 7.

2. the adaptive XTC algorithm is quite sensitive to fluctuating link quality measures.

The solution considered in this thesis to prevent a static network from unnecessary node activity are presented in this section.

6.5.1 Sensitivity of the XTC Algorithm

In practice every link quality measure shows some fluctuations, thus an important property of every algorithm is its sensitivity to inaccuracies of its inputs.

The example in figure 6.4 shows that the adaptive XTC algorithm is sensitive also to minimal fluctuations of some link quality measure. In fact, in a worst case scenario the topology control graph of the simple node constellation in figure 6.4 changes after each inquiry done by one of the network nodes.

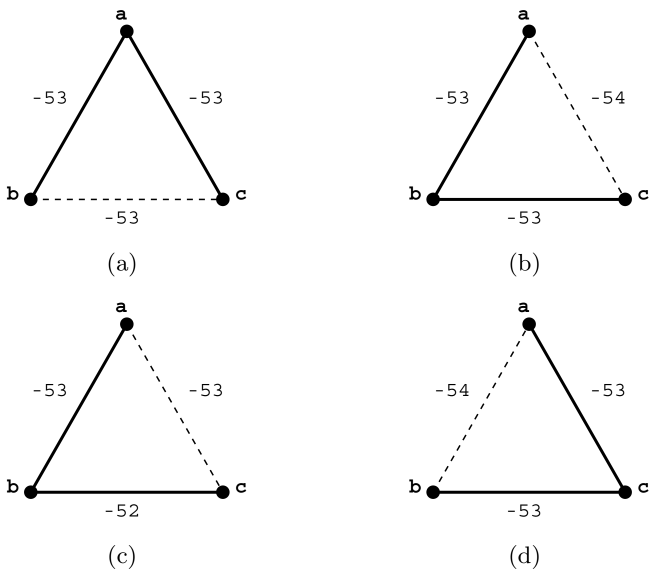


Figure 6.4: Sensitivity of the XTC algorithm: An example

6.5.2 Low pass filtering

It's clear that a more stable RSSI value can be obtained by low pass filtering the RSSI values obtained from different inquiries.

Therefore, the connection manager provides a simple IIR filter. The filter can be controlled by adjusting the filter coefficients α and β, which can be written to the EEPROM. The filter output y_k is given by

$$y_k = \frac{\alpha \cdot y_{k-1} + \beta \cdot x_k}{\alpha + \beta},$$

where y_{k-1} corresponds to the filter output of the previous step, and the input of the filter (the currently measured RSSI value) is denoted by x_k.

It's clear that by filtering RSSI values, the reaction time of the network in case of a change in topology is increased. Note that if this behavior is not desired, filtering can be turned off at any time by setting $\alpha = 1$ and $\beta = 0$ (see also section).

6.5.3 Id Mode

In the XTC algorithm, if two links are of equal quality the link to the neighbor with lower Id is considered as the "better" link. The idea of the Id mode was to establish the neighbor order in case of links with comparable link quality depending on the identities of the corresponding neighbor nodes, that is if the difference of the quality notion of two links is smaller than a predefined constant, the link corresponding to the neighbor with smaller Id will be considered as the "better" link. More precisely:

```
if ($|\omega_{uv} - \omega_{uw}| > \Delta RSSI_{id}$)
    $v \prec_u w$, if ($\omega_{uv} < \omega_{uw}$)
else
    $v \prec_u w$, if ($id_v < id_w$)
```

Thus, by setting $\Delta RSSI_{id}$ appropriately, the neighbor order won't change if links of comparable qualities are slightly fluctuating and therefore sensitivity is reduced.

Unfortunately, it turned out that the constant $\Delta RSSI_{id}$ has to be chosen very generously for that unnecessary node activity in a static network can be reduced significant. The problem of this is that by choosing $\Delta RSSI_{id}$ to big, the nodes of a neighborhood consider the link corresponding to the node with the smallest identity as the "best" link (i.e. the node with the smallest id appears in one of the first positions in the neighbor orders of its neighbors). Therefore, a lot of links will be established to nodes with a small Id, and hence the resulting network topology has "star" character rather than the desired mesh character.

6.5.4 Updating Threshold

To preserve the desirable mesh character of the resulting topology control graph and to reduce sensitivity at the same time, a further approach has been considered: instead of making the *neighbor order* robust to link weight fluctuations (which is actually done by the Id mode approach explained before), we tried to virtually stabilize the current quality of a link.

This can be done by introducing a threshold value for updating: the current link weight ω_{uv} won't be updated as long as the new link weight ω^*_{uv} measured does not change significant. More precisely:

```
if $(|\omega_{uv} - \omega^*_{uv}| > \Delta RSSI_{th})$
    $\omega_{uv} = \omega^*_{uw}$
else
    $\omega_{uv} = \omega_{uw}$
```

It's obvious that by setting the threshold $\Delta RSSI_{th}$ appropriately, the neighbor order will not change if links of comparable qualities are slightly fluctuating and therefore sensitivity is reduced.

The main advantage of this approach compared to the Id mode solution is that if some links are of comparable quality, none of the corresponding nodes is forced to be the "best" node - the sequence of slightly differing links is rather established randomly, depending on the first RSSI value measured. Therefore, the maximal degree of the resulting network is reduced compared to the maximal degree of a network making use of the Id mode solution.

6.5.5 Separated RSSI Update

As can be seen from the measurements of the RSSI done in section 7.2 (chapter 7), the fluctuations of the link weight of un-established links (RSSI value obtained from an inquiry result) are quite bigger than the fluctuations of the RSSI values read from an existing link.

Thus, another step taken to reduce sensitivity of the topology control graph was to update the link weight of connected edges separately. More precisely, instead of getting the updated link weight of an existing connection from the inquiry result, a node u reads the new link weight from the existing connection. This is done right after a neighbor discovery step (Line 3 in figure 5.1.

This leads to a more stable RSSI value of established links, and thus reduces sensitivity of the whole network.

6.6 Asymmetry

In practice, the assumption of a symmetric link quality measure does not hold. Although asymmetry of the RSSI is contained, we know from chapter 4, section 4.6 that even slightly differences in the link quality measure cancel the validity of property 2.3.2 (section 2.3) of the XTC algorithm.

To avoid this problem, we have to proceed exactly in the same way as in section 4.6: asymmetry has to be canceled from the network after each update. Thus, to cancel asymmetry due to inaccuracies of the RSSI value measured, the nodes have to balance the RSSI value measured after each update.

This can be done by simply sending the RSSI value measured with each order sent. The receiving device will then update its neighbor order according to the RSSI value measured by the sending device instead of reading the RSSI value from the existing connection.

6.7 Bounded Resources

As mentioned in the introduction, building networks with Bluetooth devices (scatternets) comes along with some restrictions:

1. A bluetooth device is not able to maintain more than ten connections at once.

2. A bluetooth device may only be a slave device of three different master devices.

Thus, if some device v is a slave device in three different piconets, some other device u won't be able to establish a connection to node v since there are not enough resources available.

In their publication [1], Wattenhofer et al. did an average-case evaluation of the bounded degree property (theorem 2.3.3), which shows that the maximal degree number ranges from 4 to 5. Therefore, by just running the adaptive XTC algorithm presented in figure 5.1, it may be that some connections cannot be established, due to the restrictions mentioned above. In this section we're going to present the approach chosen in this thesis for solving the problem of bounded resources.

6.7.1 Target

The target is to extend the XTC topology control algorithm for that the nodes of the (partial) topology control graph G_{XTC} are visible for paging devices at any time. The reason why we want to guarantee visibility of the nodes at any time is twofold.

First of all, the resulting topology control graph shall be accessible by newly appearing nodes. More precisely, a node that shall be integrated to an existing XTC network shall be able to establish a connection to each device of the network. This is needed for that new nodes can be integrated properly into an existing network.

Secondly, not ensuring visibility during network establishment, would increase the congestion problem mentioned in section 6.3 further.

Thus, to guarantee visibility of a node at any time, a slave device in three different piconets has

- either to become a master device in one of the three piconets, by doing a role switch,
- or to close the connection if no role switch can be done

as soon as possible. The situation where a role switch is not possible because resources are exhausted is shown in figure 6.5.

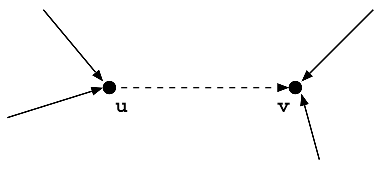


Figure 6.5: Exhausted Resources

6.7.2 Approach

The approach chosen to obtain the desired topology control graph as mentioned above can be divided into two different parts:

1. *local resource sharing during establishment*: During network establishment, the nodes change the roles if necessary and possible. Suppressed links, i.e. edges that should be in G_{XTC} but could not be kept due to exhausted resources, are marked.
2. *global resource sharing*: After network establishment, a global background process tries to distribute the available resources among the nodes, for that edges marked during network establishment can be established.

The reason for separating our solution into these two processes is twofold. First of all, the problem cannot be solved completely locally, as can be seen from the example shown in figure 6.6: assume that the links among nodes a, b, and c were established while the links among nodes u, v, and w are established at the same time. If now the link (a, u) should be established, e.g. node a must know that it is possible to switch roles with node c.

Secondly, since existing connections may be closed at a later date (e.g. due to a previously undiscovered better common neighbor), it may be that a lot of marked edges can be established during network establishment. Thus by simply marking the desired edges they can be added to G_{XTC} by local resource sharing, as well as by global resource sharing.

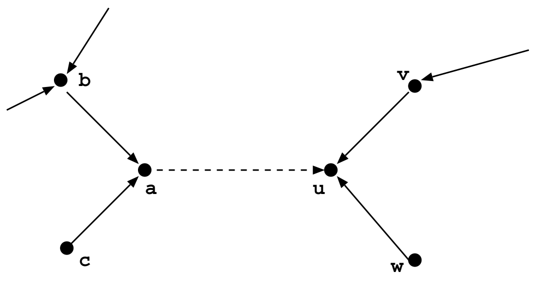


Figure 6.6: Bounded Resources: The necessity of global resource sharing

6.7.3 Local resource sharing

The idea of local resource sharing is quite simple: with each order packet sent, the sending device u includes the number of its master devices in the packet. The receiving device v may then decide if a role switch should be done or not. In fact, a role switch will only be done in one of the situations depicted in figure 6.7.

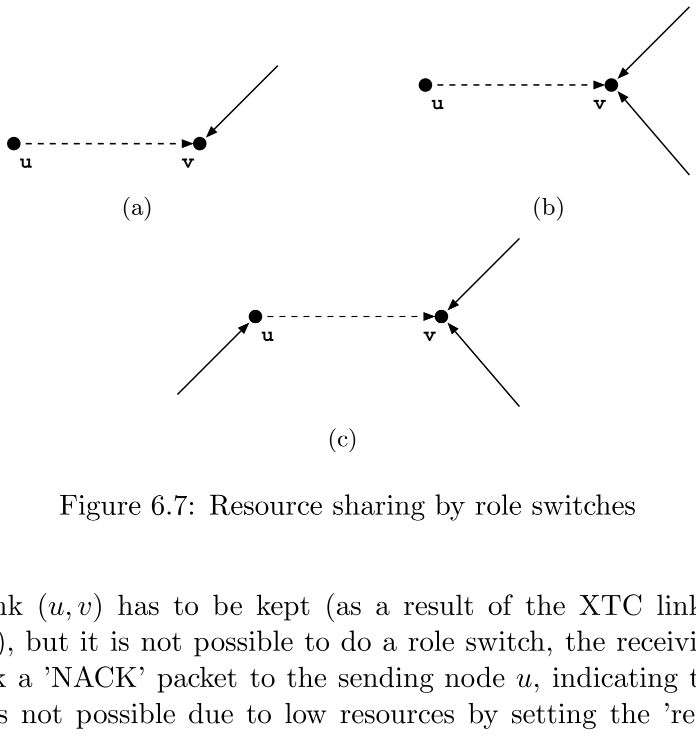


Figure 6.7: Resource sharing by role switches

If a link (u, v) has to be kept (as a result of the XTC link selection procedure), but it is not possible to do a role switch, the receiving node v sends back a 'NACK' packet to the sending node u, indicating that a role switch was not possible due to low resources by setting the 'reason' field of the 'NACK' packet to 'CM_CON_RQST_RESOURCES_EXHAUSTED' (refer to section 5.4 for details). Afterwards node u as well as node v know that the edge (u, v) should have been kept but that there were not enough resources available.

Counting Master Devices

To enable resource sharing, the devices have to count the number of master devices. This is done in the *con_table_change_cb*() callback. More precisely, from point of view of some node u, the number of master devices has to be incremented with each incoming connection. Furthermore, the number of master devices is incremented each time node u switches from master mode to slave mode.

Recovering suppressed links

If a link (u, v) cannot be kept due to low resources, nodes u and v have to mark the edge (u, v) for that they know that as soon as some resource is freed the link (u, v) has to be established since it should be in G_{XTC}. Observing resource freeing can be done in the *contable_change_cb*() by observing role switches and disconnections.

For keeping track of suppressed links, we make use of another FIFO wait queue in our implementation, to which the adjacent neighbor v of a suppressed link (u, v) can be added. Whenever a ressource because of a disconnection or a role switch is freed, an internal callback function gets called, which moves the neighbor v corresponding to the suppressed link (u, v) from this FIFO queue to the connection buffer.

6.7.4 Global Resource Sharing

The idea of global resource sharing was the following: a global background process shall share the available resources among the nodes of the topology control graph. Suppressed links are then established automatically with every resource freed, as explained before.

During verification of local resource sharing with the BTnodes, running the adaptive XTC topology control algorithm, it turned out that network establishment went fine without doing any global resource sharing, that is the resulting topology control graph contained no suppressed links in lower density networks (up to 12 nodes).

Finding an efficient mechanism for global resource sharing seems to be not that simple, and since local resource sharing seems to work fine, we did not integrate global resource sharing in our connection manager. Hence, this is something that has to be done in a future work if local resource sharing should reach its limitations in higher density networks.

6.8 Bounded Neighbor Order

In an implementation, the neighbor order will be of fixed size, since memory is limited. Therefore, another problem arises in high density networks: nodes have to be discarded from the neighbor order.

It's obvious that in a worst case scenario the resulting topology control graph may be partitioned since it is not possible for some node to determine which of the nodes in its order are necessary to guarantee connectivity. Furthermore, discarded neighbor nodes will appear in a later inquiry result again, and since a node won't have any information about the previously discarded node anymore, the node is considered as a yet unknown neighbor node again. This will considerably increase unnecessary node activity because of the successive additions / removals of neighbor nodes from the neighbor order.

Since the neighbor order can be stored quit efficiently (that is a minimum of memory is needed for storing the parameters of a neighbor node), the straight forward approach is to increase the size of the neighbor order dynamically.
But it's obvious that by increasing the neighbor order the complexity increases, that is the bigger the neighbor order, the more initial connections have to be established which increases the problem of congestion (refer to section 6.3 for further details).

The only countermeasure taken in this thesis to prevent a neighbor order overflow was by decreasing the range of the devices "by hand": we simply filtered nodes with an RSSI value bigger that $RSSI_{min}$ out of the inquiry result. The parameter $RSSI_{min}$ can be controlled by making use of the terminal, as well as by writing it to the EEPROM (see section 6.5.2).

To really solve the problem of neighbor order overflows, it would be a nice feature to let a node dynamically adjust the parameter $RSSI_{min}$ depending on the density of its neighborhood.

Chapter 7

Results

7.1 Network Establishment

In a first experiment, we randomly distributed 39 BTnodes running the adaptive XTC algorithm in the third floor of the electrical engineering building in Zurich. Target of this experiment was to get an impression of the topology graph established by a larger amount of BTnodes in a real-world indoor environment.

7.1.1 Parameters

Parameter	**Value**
$T_{p,max}$	10 sec
$T_{p,max}$	1 min
N_{init}	3
N_{trans}	4
$\sigma_{T_{inq}}$	10 sec
T_{inq}	3 sec
ai	on
$\Delta RSSI_{id}$	0
$\Delta RSSI_{th}$	8
N_{con}	2
$\sigma_{T_{con}}$	10 sec
T_{cpen}	4 sec
T_v	4 sec
α	1
β	3

7.1.2 Results

The automatically established network together with the floor plan is shown in figure 7.1, the network graph obtained from the JAWS GUI is shown in figure 7.2.

As expected, the graph is connected, and the shortest cycles are of length 4. The mean degree is very low ($\approx 1,2$), but also the maximal degree of the network is only 5. Furthermore, there are no excessive, power-consuming long-distance links.

Visibility

A nice view of the established network is given in figure 7.3, which was also obtained from the JAWS GUI: not only the links established (bold lines) are shown, but also the links of the unit disk graph (remaining lines).

Note that from figure 7.3 the solid walls separating the two corridors are clearly identifiable: nodes positioned on opposite floors do not see each other - except nodes 00:f9 and 00:33 which is reasonable since they are only separated by an air space.

Resource Sharing

Note that the line connecting nodes 00:43 and 00:69 in figure 7.3 does not correspond to an existing link: this is a link that could not be establishes because of exhausted resources (refer to section 6.7 for details) during network establishment. In fact, if the link would be established, the corresponding nodes would close it immediately again since the links (00:d9, 00:43) and (00:d9, 00:69) are preferred.

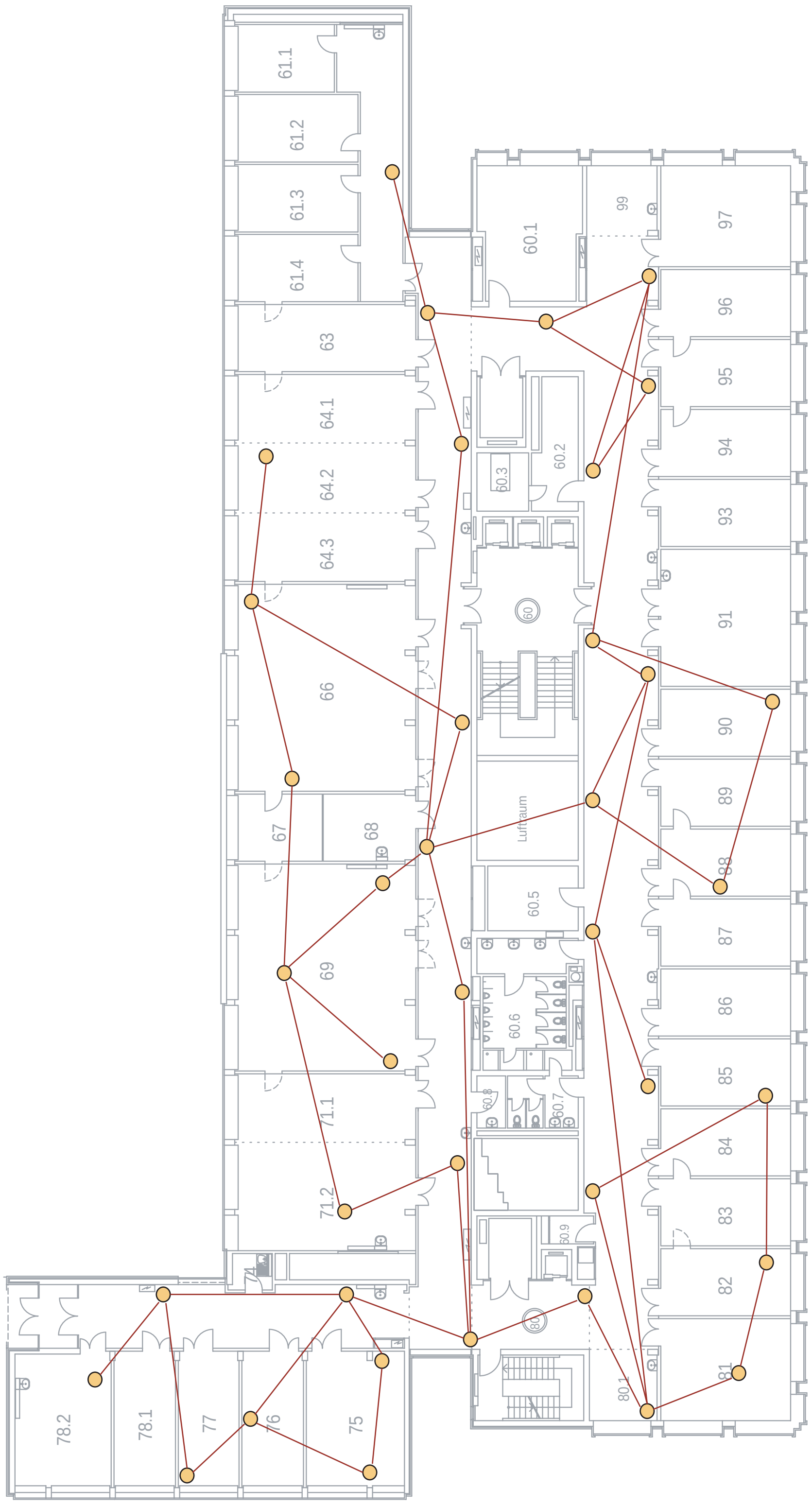


Figure 7.1: Network established by distributing 39 BTnodes randomly in an indoor environment

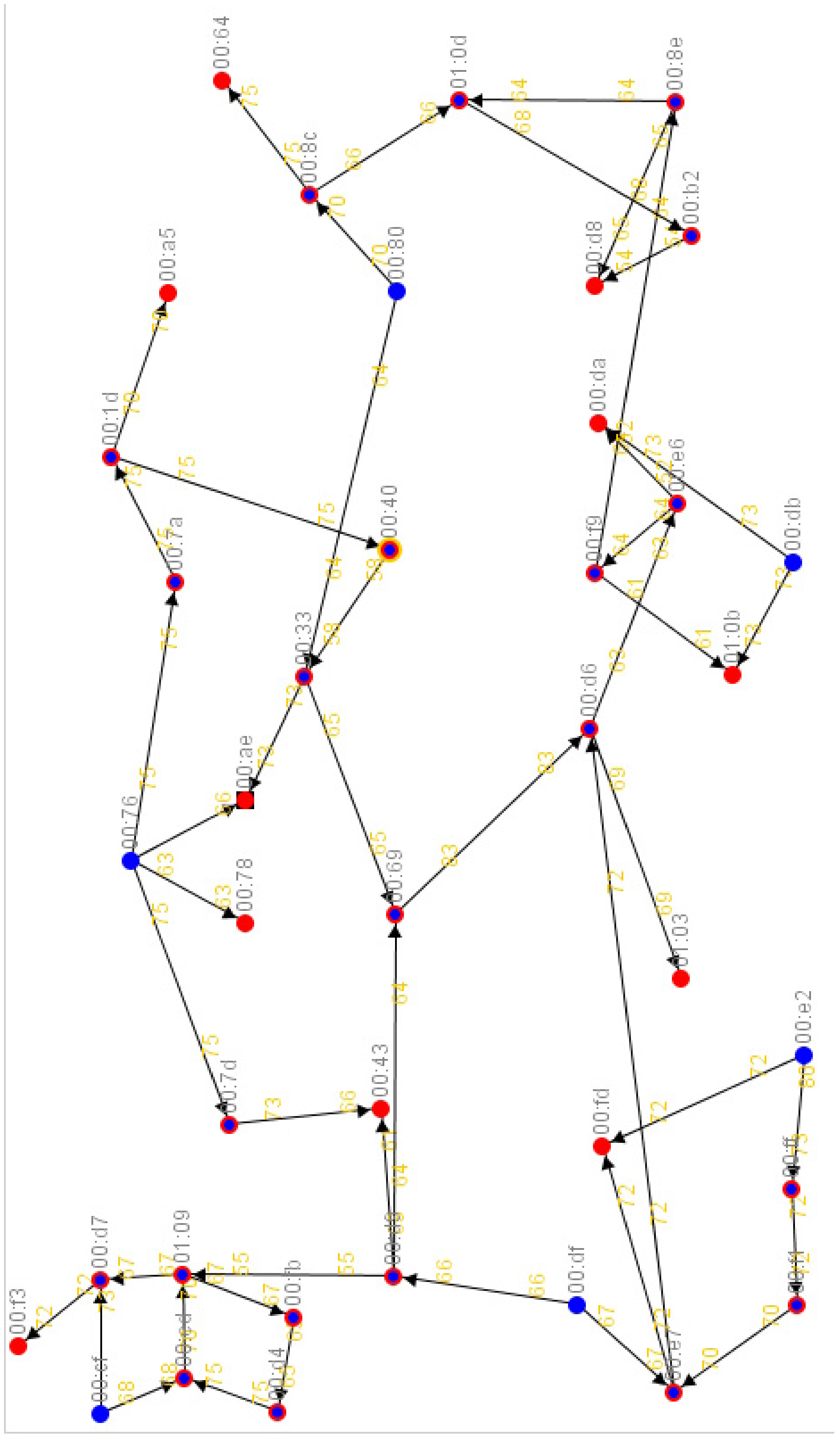


Figure 7.2: JAWS GUI representation of the network depicted in figure 7.1

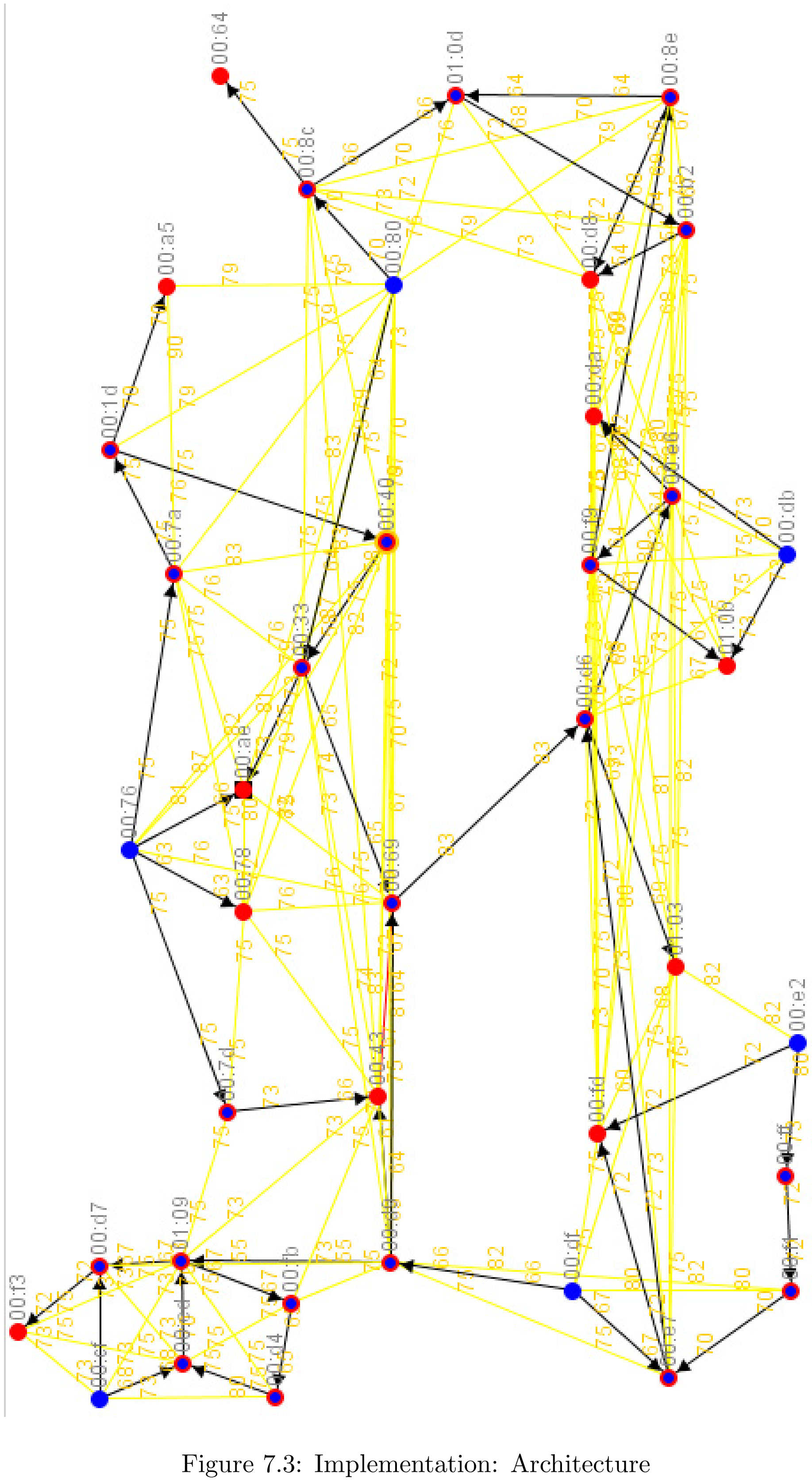


Figure 7.3: Implementation: Architecture

7.2 RSSI Measurement

In this measurement, we've measured the RSSI value against the distance obtained from two BTnodes that were in line of sight. The RSSI value was measured in two different ways: in a first step we've measured the RSSI value obtained from an inquiry, and in a second step we've measured the RSSI obtained from an established connection. For each distance we've repeated the measurement ten times.

7.2.1 Results

The results are shown in figure 7.4. Each diagram shows the RSSI value measured against the distance of the two nodes. The boxes indicate the deviation of the mean value, which is marked by the dash inside of a box. Minimal and maximal values measured are marked by a small "T".

Considering only the mean values of the plots, the result is quite remarkable. Although not a straight line, it's observable that the RSSI value decreases with increasing distance as desired.

7.2.2 Conclusion

The deviation, as well as the minimal (maximal resp.) values measured indicate that the RSSI value is extremely fluctuating when getting it from an inquiry. Thus, by obtaining the RSSI value from a single inquiry result, it's not possible to get information about the distance between two devices. Or, put the other way round, it's hard to position BTnodes in a way that RSSI value differences obtained are significant.

Another problem is obvious: since the XTC algorithm is quite sensitive to fluctuating link quality measures (refer to section 6.5), it's impossible to stabilize even a static network.

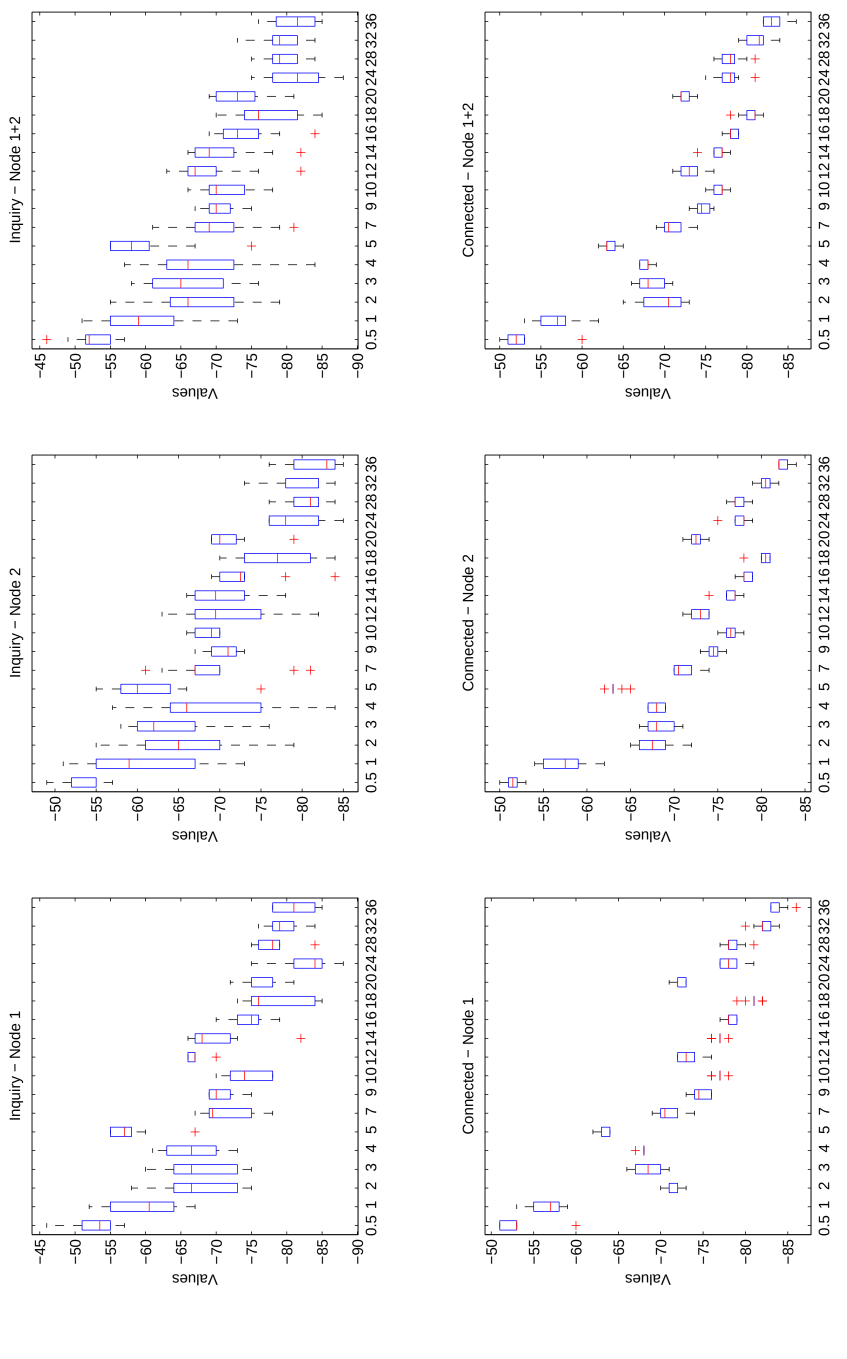


Figure 7.4: RSSI measurements

7.3 Reducing Sensitivity

In this section, we're going to show that the approaches presented in section 6.5 do solve the problem of sensitivity but each of it comes along with disadvantages.

7.3.1 RSSI Threshold

In a first experiment, we've investigated how big the updating threshold $\Delta RSSI_{th}$ has to be chosen for that the resulting network topology of a static node constellation remains stable, while the IIR filter was disabled.

This was done by placing 12 BTnodes randomly in the students laboratory, turning them on, and counting the number of total connection establishments during 20 minutes. This was repeated for four different values of $\Delta RSSI_{th}$. The parameter settings are given in the table below.

Parameter	**Value**
$T_{p,max}$	10 sec
$T_{p,max}$	1 min
N_{init}	3
N_{trans}	4
$\sigma_{T_{inq}}$	10 sec
T_{inq}	3 sec
ai	on
$\Delta RSSI_{id}$	0
$\Delta RSSI_{th}$	varying
N_{con}	2
$\sigma_{T_{con}}$	10 sec
T_{cpen}	4 sec
T_v	4 sec
α	1
β	0

Results

The results are shown in figure 7.5. It can be seen that for small values of $\Delta RSSI_{th}$, the network won't stop adjusting its topology due to the fluctuating RSSI values measured. It turns out that the threshold $\Delta RSSI_{th}$ has to be chosen quite generously for that the topology of the network remains stable. In fact, the network topology seems to be stable not until $\Delta RSSI_{th} \geq 16$.

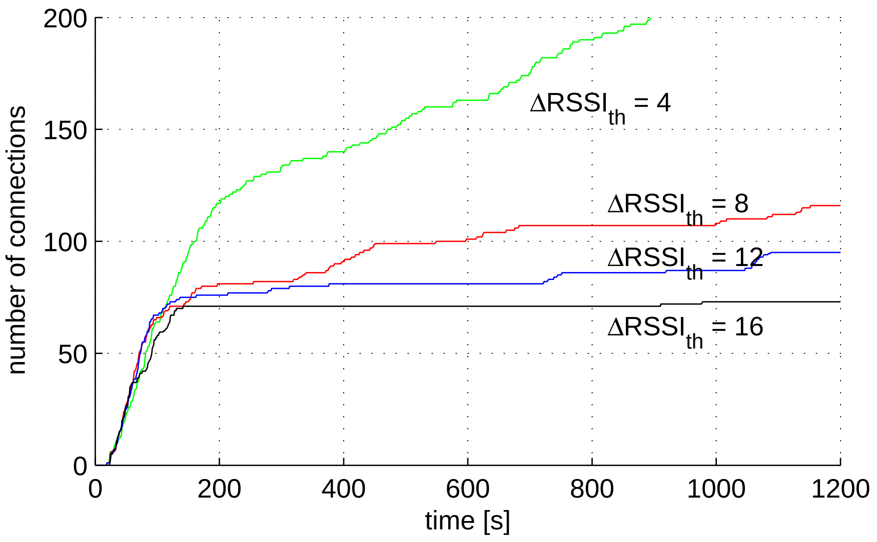


Figure 7.5: Number of total connection establishments against time, for different values of $\Delta RSSI_{th}$. Low pass filter is off.

Conclusion

The problem is obvious: if we choose $\Delta RSSI_{th} \approx 16$, we dramatically quantize the possible link qualities: In fact, a node won't be able to distinguish more than approximately three different link qualities[1], that is, from point of view of some node u, the quality ω_{uv} of some link (u, v) is either "good", "medium", or "bad".

Hence, the idea of adapting the network topology due to changing link qualities to conserve energy slightly depreciates. But still, "bad" links (i.e. long-distance links) can be avoided if there is a better neighbor to which at least a link of "medium" quality exists.

7.3.2 Filtering

This measurement is identical to the measrement done in the previous subsection, that is we again measured the number of total connection establishments against the RSSI threshold $\Delta RSSI_{th}$. But this time we filtered the RSSI values measured by setting $\alpha = 1$, and $\beta = 3$. The parameter setting is given in the table below.

[1]if we assume that the maximal RSSI value that can be measured is ≈ -45 and the maximal value ≈ -85

Parameter	Value
$T_{p,max}$	10 sec
$T_{p,max}$	1 min
N_{init}	3
N_{trans}	4
$\sigma_{T_{inq}}$	10 sec
T_{inq}	3 sec
ai	on
$\Delta RSSI_{id}$	0
$\Delta RSSI_{th}$	varying
N_{con}	2
$\sigma_{T_{con}}$	10 sec
T_{cpen}	4 sec
T_v	4 sec
α	1
β	3

Results

The results of the measurement are shown in figure 7.6. In contrast to the previous measurement, the network topology remains constant by choosing $\Delta RSSI_{th} \approx 8$. Note that it takes quite a long time until the IIR filter is in steady state (up to 6-7min).

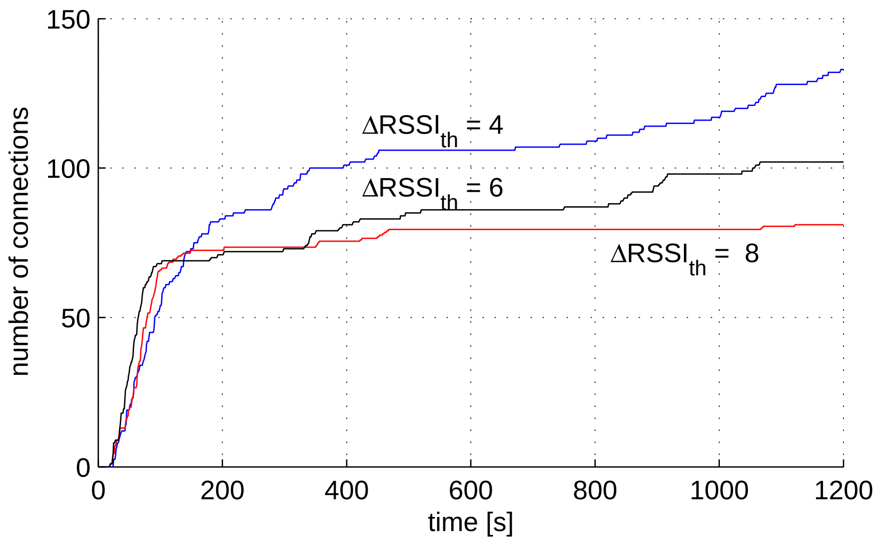


Figure 7.6: Random inquiry timing

Conclusion

By low pass filtering the RSSI values obtained from the inquiry results, the number of distinguishable link qualities can be slightly increased. More precisely, by assuming the RSSI values measured to be in $[-45, -85]$, it is possible to distinguish approximately five different link qualities.

7.4 Network Setup Time

In this section we're going to show that the relaxed timing scheme explained in section 6.4 is necessary for that the BTnodes are able to establish connections to each of their neighbor nodes if node density is high. Additionally, the network setup time was measured against node density.

7.4.1 Relaxed Timing

In this measurement, we've measured the network setup time in a higher density network against the maximal connection establishment period $\sigma_{T_{con}}$. This has been done by placing 15 Btnodes randomly on a table, and starting them with a synchronous reset. For each $\sigma_{T_{con}}$ the setup time was measured three times. The target was to show that by the applying the relaxed timing scheme, the problem of congestion can be solved.

Parameter	**Value**
$T_{p,max}$	10 sec
$T_{p,max}$	1 min
N_{init}	3
N_{trans}	4
$\sigma_{T_{inq}}$	10 sec
T_{inq}	3 sec
ai	on
$\Delta RSSI_{id}$	0
$\Delta RSSI_{th}$	90
N_{con}	2
$\sigma_{T_{con}}$	varying
T_{cpen}	4 sec
T_v	4 sec
α	1
β	0

Results

In figure 7.7, the averaged network setup time against $\sigma_{T_{con}}$ is depicted. By choosing $\sigma_{T_{con}}$ between 6 to 10 seconds, everything is fine, and the network establishment time is around 200 seconds. But by decreasing $\sigma_{T_{con}}$, network establishment time increases dramatically up to ≈300 to 400 seconds.

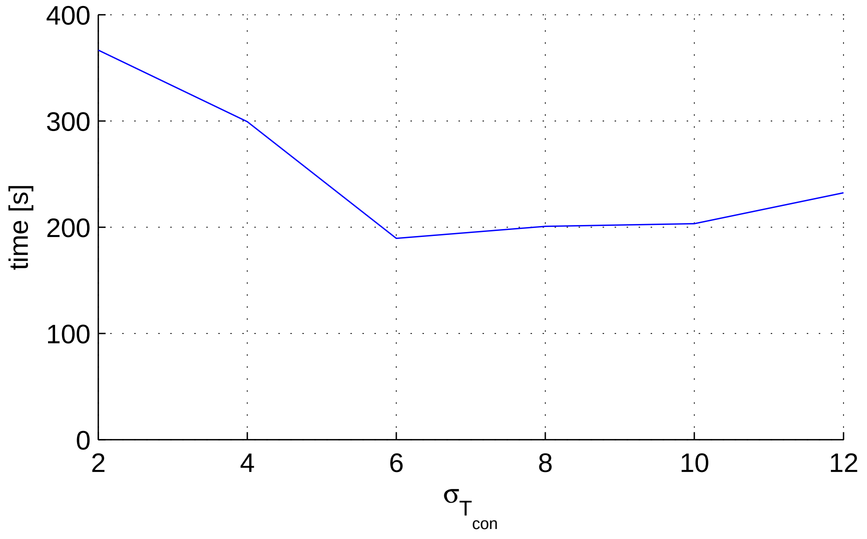


Figure 7.7: Network establishment time against $\sigma_{T_{con}}$ with a node density of 15.

Conclusion

This measurement shows that by choosing $\sigma_{T_{con}}$ to low, network establishment increases dramatically because of congestion.

7.4.2 Setup Time Measurement

In this measurement, we've measured the network setup time against node density. This has been done by placing Btnodes randomly on a table, and starting them with a synchronous reset. For that it was possible that each BTnode was able to establish a connection to each neighbor device, $\sigma_{T_{con}}$ had to be set to 20 sec[2].

[2]when decreasing $\sigma_{T_{con}}$ the nodes were not able to do all of the necessary connection establishments. In fact, strange behavior of the nodes has been observed in a network with a density of 18, which seems to trace back to an error in either the BT Stack or the Bluetooth module.

Parameter	Value
$T_{p,max}$	10 sec
$T_{p,max}$	1 min
N_{init}	3
N_{trans}	4
$\sigma_{T_{inq}}$	10 sec
T_{inq}	3 sec
ai	on
$\Delta RSSI_{id}$	0
$\Delta RSSI_{th}$	90
N_{con}	2
$\sigma_{T_{con}}$	20 sec
T_{cpen}	4 sec
T_v	4 sec
α	1
β	0

Results

In figure 7.8, the averaged network setup time against $\sigma_{T_{con}}$ is depicted. Instead of being a linear function of density, the averaged network setup time seems to have exponential character. The reason for this may be that the probability of paging collisions increases with increasing node density.

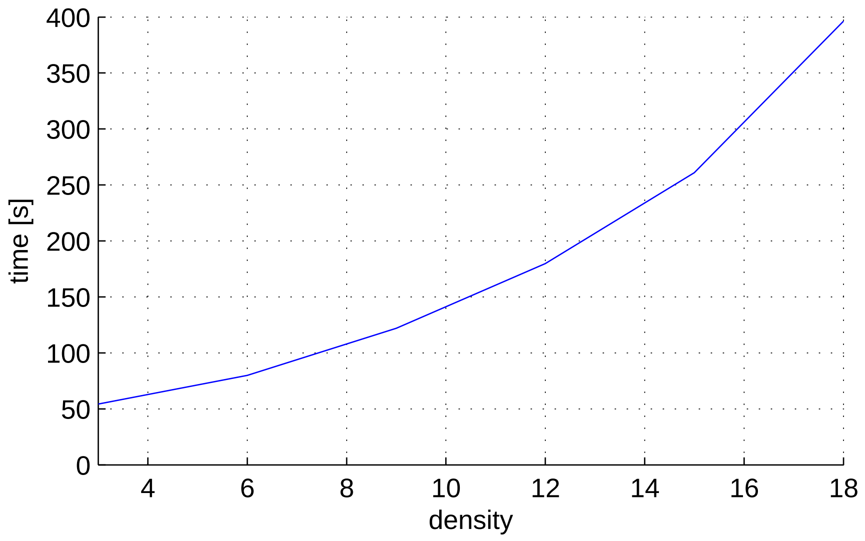


Figure 7.8: Network establishment time against node density, with $\sigma_{T_{con}} = 20 sec$

Conclusion

As can be seen from figure 7.8, the performance of network establishment is rather poor, especially in higher density networks. Obviously, to improve the network setup time in lower density networks, $\sigma_{T_{con}}$ can be decreased.

But with decreasing $\sigma_{T_{con}}$ network setup time in higher density networks increases. Thus, to improve performance of network establishment further, it seems to be necessary to make approximations, that is a node does only establish connections to a predefined number of neighbor nodes. But the problem of such approximations is obvious: in a worst case scenario, the resulting topology control graph may be partitioned.

Chapter 8

Conclusion

In this thesis, an adaptive variant of the XTC algorithm has been developed and implemented on the BTnode. It was shown that the implemented algorithm has successfully established the XTC topology control graph within a larger deployment of 39 BTnodes in an indoor environment.

It was shown that the strict locality of the XTC algorithm gives the BTnodes a lot of trouble during network establishment in higher density networks. To solve this problem, a relaxed timing scheme has been applied, which enabled correct network establishment with up to 18 BTnodes. Nevertheless, the implementation performs bad during network establishment in higher density networks. Thus, to improve the performance in higher density further, approximations are inevitable.

Furthermore, in higher density networks, the common limitations in Bluetooth scatternets come into play. Hence, the available resources are shared locally to increase the degree of the correctness achieved, but this does not completely solve the problem - to guarantee the correctness of the resulting topology control graph, the implementation would have to be extended by a global resource sharing algorithm.

When testing the ability of dynamic adaption, it turned out that dynamic adaption led to excessive node activity even in a static deployment and became rather a problem than a desired feature. The reason for this behavior are the extremely fluctuating link quality measure provided by the BTnode, as well as the high sensitivity of the XTC algorithm. By low pass filtering the link quality measure obtained and by introducing an updating threshold, unnecessary node activity could be reduced. As a result, a BTnode is able to distinguish approximately five different link quality intervals.

Last, but not least, the XTC algorithm looses one of its key properties in case of asymmetric link qualities, that is it cannot be guaranteed that cycles of girth 3 are avoided. In this thesis, asymmetry was successfully canceled by continuously calibrating the link weights measured.

Hence, although being one of the most simple topology control algorithm available, a lot of engineering efforts had to be taken in order to obtain correctness of the resulting topology control graph. In fact, compared to the 10 lines of pseudo code that were used by Wattenhofer at al. to formulate the XTC algorithm, the implementation of the adaptive XTC algorithm presented in this thesis ended up in approximately 2000 lines of C code (not including the various library and system functions used, as well as the HCI Interface functions that are provided by the BT Stack).

Appendix A

Terminal Commands

All terminal commands provided by the connection manager are executed by typing “cm” plus the desired command connection manager command. An overview is given in the table below.

Command	**Parameters**	**Description**
inq	-	does an inquiry of duration T_{inq}, writes the inquiry result to the ’inqres’ struct, and updates the neighbor order accordingly.
inqres	-	prints out the latest inquiry result.
neighbors	-	prints out the neighbor order.
con	<index>	establishes a connection to the neighbor that is at position *index* in the current neighbor order, and sends the current order to it.
check	<index>	sends the current neighbor order to the neighbor with index *index*
conbuf	-	prints out the neighbors contained in the ’con_buf’
sendbuf	-	prints out the neighbors currently contained in the ’send_buf’
blocked	-	prints out the currently blocked neighbors, that is neighbors to which a XTC link should be established but there are not enough resorces available.
masters	-	prints out the current master devices
rc	-	prints out the connection handles of the currently established *reliable* connections

Command	**Parameters**	**Description**
autinq	[0|1]	turns on/off automatic inquiry
debug	[0|1]	turns on/off debug mode
aitime	<value>	sets the duration T_{inq} of an inquiry to *value* seconds
aimin	<value>	sets the minimal inquiry period $T_{p,min}$ to *value* seconds (refer to section 5.2.4)
aimax	<value>	sets the maximal inquiry period $T_{p,max}$ to *value* minutes (refer to section 5.2.4)
aistart	<value>	sets the number of initial inquiries N_{init} to *value* (refer to section 5.2.4)
aitrans	<value>	sets the number of transitional inquiries N_{trans} to *value* (refer to section 5.2.4)
aidev	<value>	sets the maximal inquiry period deviation $\sigma_{T_{inq}}$ to *value* seconds (refer to section 6.1)
rssi	<value>	sets the updating threshold $\Delta RSSI_{th}$ to *value* (refer to section 6.5).
idmode	<value>	sets the minimal RSSI difference $\Delta RSSI_{id}$ needed for that two different link qualities are not considered as being equal to *value* (refer to section 6.5).
cperiod	<value>	sets the maximal wait time $\sigma_{T_{con}}$ between two connection establishments to *value* seconds (refer to section 6.3).
cerradd	<value>	sets the wait time T_{cpen} after a page collision occurred to *value* seconds (refer to section 6.2).
cvis	<value>	sets the minimal visibility time T_v to *value* seconds (refer to section 6.3).
ctries	<value>	sets the maximal number of connection trials N_{con} to *value* (refer to section 6.4).
ptime	<value>	writes the page timeout to the EEPROM. After restart, the page timout wil be set to *value* seconds.

Command	**Parameters**	**Description**
lpfa	<value>	sets coefficient α of the low pass IIR filter (refer to section 6.5.2).
lpfb	<value>	sets coefficient β of the low pass IIR filter (refer to section 6.5.2).
maxneighs	<value>	writes the maximal size of the neighbor order to the EEPROM. After restart, the maximal size of the neighbor order is set to *value* (refer to section 5.2).
inqdevs	<value>	writes the maximal size of the inquiry result to the EEPROM. After restart, maximal number of devices some node u is going to look for during inquiring *value* (refer to section 5.2).
eepreset	-	Sets the parameters in the EEPROM to its default values.

Appendix B

EEPROM Parameters

Most of the parameters explained in section 5.3 can be written to the EEPROM. Setting a parameter that is stored in the EEPROM can be done with the terminal using the syntax:

cm <cmd> <value> 1

where <cmd> is the command to set the desired parameter, <value> is the according value, and the terminating '1' ensures that the value passed is written to the EEPROM.

To reset the EEPROM to its default values, use the terminal command *eepreset* (refer to appendix A. The following table gives an overview of the parameters that are stored in the EEPROM, its default values, and the corresponding terminal command.

Parameter	**Default Value**	**Command**
T_{inq}	3 sec	aitime
$T_{p,min}$	10 sec	aimin
$T_{p,max}$	1 min	aimax
N_{init}	3	aistart
N_{trans}	4	aitrans
$\sigma_{T_{inq}}$	10 sec	aidev
$\Delta RSSI_{th}$	12	rssi
$\Delta RSSI_{id}$	0	idmode
$\sigma_{T_{con}}$	10 sec	cperiod
T_{cpen}	4 sec	cerradd
T_v	4 sec	cvis
N_{con}	2	ctries
t_p	7 sec	ptime
α	1	lpfa
β	3	lpfb
N_{neighs}	15	maxneighs
N_{inq}	15	inqdevs

Appendix C

Problem Task

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Sommersemester 2004

MASTERARBEIT MA-2005.05
für

Kevin Martin

Tutor:	Matthias Dyer	(dyer@tik.ee.ethz.ch)
Co-Tutor:	Jan Beutel	(beutel@tik.ee.ethz.ch)
Professor:	L. Thiele	

Ausgabe: 30. November, 2005
Abgabe: 30. Mai, 2005

JAWS - Scatternets with BTnut

Introduction

Figure 1: BTnode rev.3

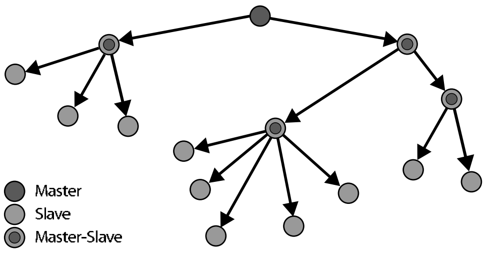


Figure 2: Example of a Bluetooth Scatternet

A sensor network is a collection of small, low-resource devices that are distributed in the physical environment. Due to cost and flexibility issues, it is often assumed to be a wireless sensor network (WSN) consisting of a large number of sensor nodes. Each of these nodes collects sensor data, and the network collaboratively provides high-level sensing results.

One key challenge of sensor networks is the formation and maintenance of a connected network that provides a reliable data transport. To form such a network with Bluetooth implies the formation of *Scatternets* [6] (Fig. 2. The recent increase in research interest has led to many new algorithms [5, 7] for the formation of Bluetooth Scatternets. However, very few have been implemented and tested on real devices.

The *BTnode rev.3* [1] (see Fig.1) is a very recent plattform for the development of sensor–network applications and protocols. It has two radio interfaces: a Bluetooth radio provides relatively high bandwith, while the second radio is for low-rate and low-power operation. Recently, a new system software [2] for the BTnode rev.3 has been released, that is based on the Ethernut embedded OS [3].

JAWS [4] is an application initially designed for the older BTnode rev.2, that has now been ported for the BTnode rev.3. It contains a straight–forward Scatternet–formation algorithm. The BTnodes running JAWS form automatically to a tree network. This network is further used to provide virtual serial connections. Due to the

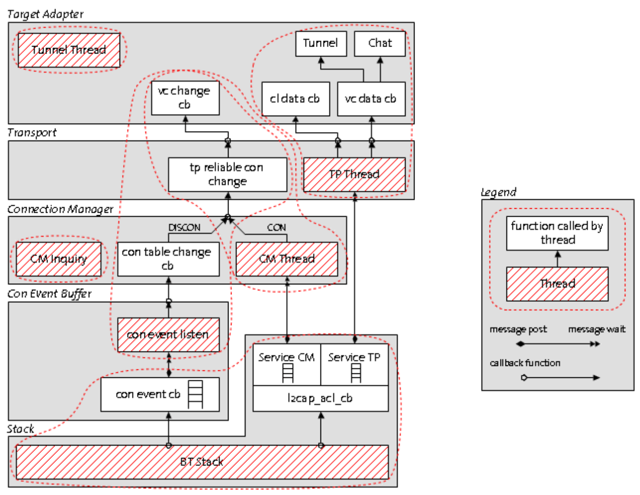


Figure 3: JAWS Components

modular structure of JAWS (Fig. 3), the *Connection Manager* component, which implements the algorithm can be independently replaced.

The current implementation has following limitations:

- The topology formed is a tree. There is only one route from a source to its destination. If this link fails, this destination (and the whole subtree) are disconnected from the network.
- The connection manager algorithm implies that all devices periodically inquire for device discovery. However, an inquiry on the new BTnode hardware suspends all data transport for the duration of the inquiry (typically 2-4 sec.).

XTC [7] is a simple, yet effective algorithm, that can be used for Scatternet formation and that is targeted to resource constrained nodes such as the BTnode.

Problem task

The goal of the project at hand is to evaluate different Scatternet formation algorithms, to select one for implementation on the BTnodes and to obtain qualitative and quantitative measures on the performance.

Teilaufgaben

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den relevanten Arbeiten im Bereich Sensornetze und Scatternets formation vertraut. Führen Sie eine Literaturrecherche durch. Suchen Sie auch nach relevanten neueren Publikationen.
3. Arbeiten Sie sich in die Softwareentwicklungsumgebung der BTnodes ein. Machen Sie sich mit den erforderlichen Tools vertraut und benutzen Sie die entsprechenden Hilfsmittel (online Dokumentation, Mailinglisten, Application Notes).

4. Machen Sie sich mit der JAWS Applikation vertraut. Schauen Sie sich insbesondere die Schnittstelle zum Connection Manager an. Arbeiten Sie sich in die Grundlagen von Bluetooth ein. Wesentlich für diese Arbeit ist vor allem das *HCI Interface* und alles was das *Device Discovery* und *Connect* betrifft.
5. Implementieren Sie einen ausgewählten Scatternet formation algorithm auf den BTnodes rev.3. Testen Sie Ihre Implementierung mit einem Testbed von 10–30 BTnodes.
6. Definieren und messen sie relevante Charakterisiken Ihrer Implementierung. Führen Sie ein systematisches Parameter-Tuning durch.
7. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Semesterarbeit

Allgemeines

- Der Verlauf des Projektes Semesterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- Sie verfügen über PC's mit Linux/Windows für Softwareentwicklung und Test. Für die Einhaltung der geltenden Sicherheitsrichtlinien der ETH Zürich sind Sie selbst verantwortlich. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Semesterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern. Verfassen Sie dazu auch einen kurzen wöchentlichen Statusbericht (EMail).

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *30. Mai 2005* dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.

References

[1] Btnodes, a distributed environment for prototyping ad hoc networks. http://www.btnode.ethz.ch.

[2] Btnut system software reference. http://www.btnode.ethz.ch/support/btnut_api/index.html.

[3] Ethernut. http://www.ethernut.de/.

[4] Jaws, deployment-support networks for next generation prototyping of sensor networks. http://www.btnode.ethz.ch/projects/jaws/jaws_api/.

[5] S. Basagni, R. Bruno, and C. Petrioli. A performance comparison of scatternet formation protocols for networks of Bluetooth devices. In *Proc. 1st IEEE Int'l Conf. Pervasive Computing and Communications (PerCom 2003)*, pages 341–350. IEEE CS Press, Los Alamitos, CA, March 2003.

[6] Bluetooth Special Interest Group. *Specification of the Bluetooth System - Core, v.1.2*, March 2003.

[7] Roger Wattenhofer and Aaron Zollinger. XTC: A practical topology control algorithm for ad-hoc networks. In *4th International Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN)*, Santa Fe, New Mexico, April 2004.

Bibliography

[1] R. Wattenhofer and A. Zollinger. XTC: A Practical Topology Control Algorithm for Ad-Hoc Networks. In *Proc. of the* 18th *Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[2] Bluetooth specification and additional information.
http://www.Bluetooth.org

[3] Btnodes, a distributed environment for prototyping ad-hoc networks.
http://www.btnode.ethz.ch

[4] Nut OS main page.
http://www.ethernut.de/en/software.html

[5] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks.
http://btnode.ethz.ch/projects/jaws

[6] JAWS Application Documentation.
http://www.btnode.ethz.ch/projects/jaws/jaws_api/main.html

[7] J. Beutel, M. Dyer, L. Meier, and L. Thiele: Scalable Topology Control for Deployment-Support Networks. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05), pages 359-363*, April 2005.

[8] BTnut System Software.
http://www.btnode.ethz.ch/support/btnut_api/index.html

[9] S. Basagni, R. Bruno, G. Mambrini, and C. Petri-oli, Comparative performance evaluation of scatter-net formation protocols for networks of Bluetooth de-vices, In *Wireless Networks, vol. 10, no. 2, pp. 197 213*, March 2004.