KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

NanoTorrent: Locality-aware peer-to-peer file distribution in wireless sensor networks

Mattias Buelens

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Gedistribueerde systemen

Promotor:
Prof. dr. Danny Hughes

Assessoren:
Dr. Sam Michiels
Dr. ir. Jan Ramon

Begeleiders:
Dr. Nelson Matthys
Wilfried Daniels

Academiejaar 2014 – 2015

Preface

I would like to thank my coordinators Nelson Matthys and Wilfried Daniels, for guiding me throughout the year to make this thesis. They provided me with valuable insights into the many aspects and challenges of Internet of Things and wireless sensor networks. They were always very helpful whenever I had questions, and always asked the right questions to make me better understand potential problems and come up with solutions.

I would also like to thank my promotor prof. Danny Hughes, for making this thesis possible and coming up with great ideas to improve the protocol design.

Finally, I want to thank you – the reader – for taking the time to read my thesis text. Whether you're a proofreader, a jury member or you just thought this thesis might be worth a look, I appreciate the effort and I'd love to know your feedback.

Mattias Buelens

Contents

Abstract

Wireless sensor networks consist of many small computer devices equipped with various sensors and actuators which communicate with each other using their wireless antenna. These networks are a key part of the Internet of Things vision, where these devices are deployed to monitor or control real-life physical things and make them 'smart' by connecting them to the Internet. In order to keep wireless sensor networks operating for long periods of time, they must be able to adapt and evolve even after many years after being deployed. This requires them to retrieve large files such as new programs or configurations over the network fast and efficiently.

NanoTorrent is a peer-to-peer protocol that allows fast distribution of files in wireless sensor networks. It introduces the use of a hybrid peer discovery mechanism with both a tracker and local neighbour discovery, allowing it to work even in heterogeneous deployments consisting of different types of nodes running different programs. The peer-to-peer approach helps balance the load of the file distribution on the seed and the network, and takes advantage of link-local multicast messages to distribute parts of the file to many neighbours simultaneously.

The evaluation of the protocol shows that NanoTorrent can provide fast file distribution in many different network configurations. The distribution speed scales well with the size of the network, although in its current form the amount of transmissions is still an issue. The hybrid peer discovery approach means that peers send a lot of messages to discover other peers, and can find distant peers which require messages to take multiple hops across the network to reach them. More research and experimentation is needed to reduce the needed communications.

List of Figures

List of Terms and Abbreviations

List of Terms

initial seed

Seed that bootstraps the torrent's distribution. This seed has not downloaded the file from other peers, but has been set up by the distributor with the entire file already available.

leecher

Peer which are still missing parts of the distributed file.

peer

A client participating in a torrent's distribution.

seed

Peer which has finished downloading the entire distributed file.

seeder

See: seed

swarm

The active peers of a torrent.

torrent descriptor

Metadata about a torrent, consisting of the location of the tracker (its IPv6 address) and the torrent information.

torrent info hash

SHA-1 hash of the torrent information. This hash acts as an identifier for the torrent, used in messages by trackers and peers to identify the torrent.

torrent information

Metadata describing the file to be distributed by a torrent. This consists of the file's size, the number of pieces that make up the whole file, the size of each piece and the SHA-1 hashes of all pieces.

tracker

Server tracking the state of the swarm of one or more torrents, allowing clients to discover other peers in the swarm.

List of Abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
CFS	Contiki File System
DDoS	distributed denial of service attack
DHCP	Dynamic Host Configuration Protocol
EEPROM	electrically erasable programmable read-only memory
FTP	File Transfer Protocol
GB	gigabyte (10^9 bytes)
GHz	gigahertz (10^{-9} seconds)
HTTP	Hypertext Transfer Protocol
ICMPv6	Internet Control Message Protocol version 6
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	internet service provider
IT	information technology
JNI	Java Native Interface
KB	kilobyte (10^3 bytes)
LAN	local area network
MAC	media access control
MHz	megahertz (10^{-6} seconds)
MTU	maximum transmission unit
P2P	peer-to-peer
RAM	random access memory
RPL	IPv6 Routing Protocol for LLNs
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WSN	wireless sensor network

Chapter 1

Introduction

Wireless sensor networks (WSNs) enable scientists to monitor active volcanoes without having to risk their lives when they need seismic measurements [17], make homes smarter by connecting all appliances together [2] and will allow future energy production to be tuned to the real-time energy demand of consumers [13] [14]. However, WSNs need to evolve in order to last for many years, and thus operators must be able to distribute updates to nodes in these networks [3] [25]. Since these nodes are very resource-constrained and often battery-powered (e.g. AVR Zigduino [7], TMote Sky [5]), the distribution of such large update files must be done in an efficient but fast manner.

This master's thesis presents NanoTorrent as a peer-to-peer (P2P) file distribution protocol aimed at WSNs. The problem context is described in section 1.1, going in-depth about the Internet of Things and wireless sensor networks and explaining peer-to-peer architectures. Section 1.2 discusses the problem and challenges imposed by WSNs. Section 1.3 sketches the contributions of the proposed solution. Finally, section 1.4 outlines the structure of the text.

1.1 Problem context

1.1.1 Internet of Things

The Internet of Things (IoT) is a vision where many small embedded electronic devices are deployed in a physical environment to monitor their environment and communicate with each other and the outside world to achieve certain goals [24]. In the research context for Internet of Things (IoT), the challenge is to make the physical world 'smart': physical objects become connected to the Internet and can be monitored or operated from anywhere at any time. In other words, bringing the world of 'things' to the digital world of the Internet, as shown in figure 1.1.

In order to integrate such a large number of devices into the global Internet infrastructure, IoT devices use IPv6 for networking. The IPv6 address space allows for 2^{128} possible addresses, which is more than plenty to support the rapid growth of these devices. The same technologies that power the Internet are adapted to the scale of IoT devices, such that they can communicate with outside networks in a

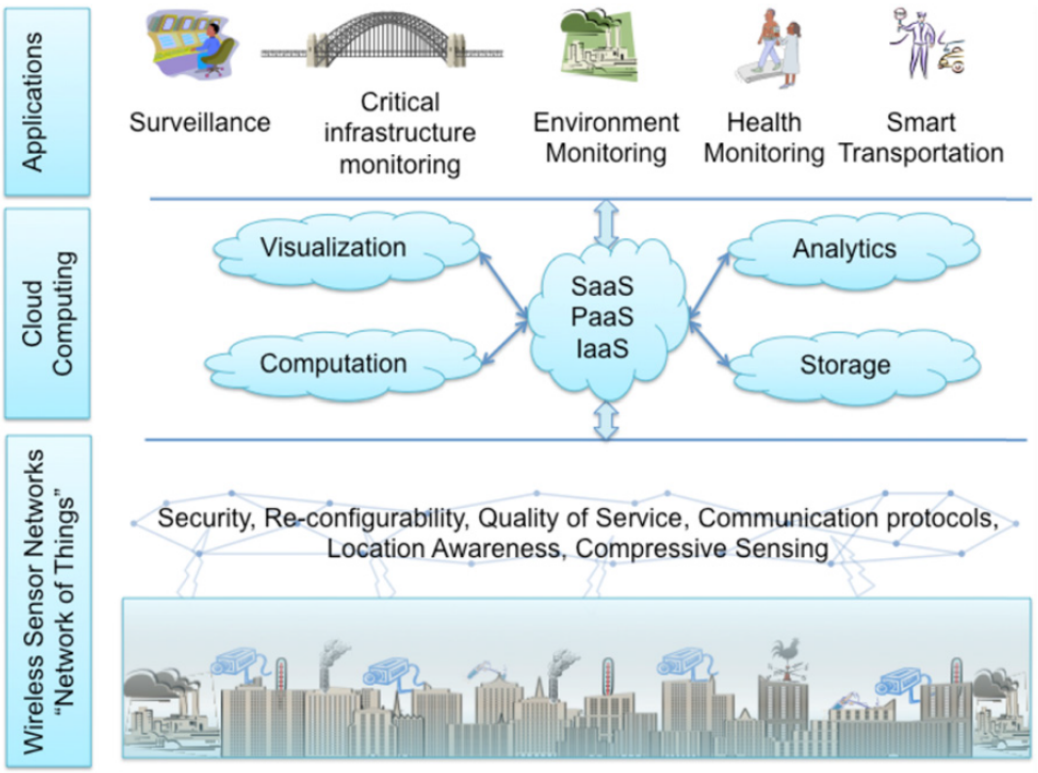


Figure 1.1: A conceptual framework for IoT applications, integrating the world of 'things' (in WSNs) with the digital world of the Internet. [12]

uniform and transparent manner. For example, IoT devices can send and receive IPv6 packets, download a file from a website, or even host a small web server.

The (possible) applications for IoT encompass nearly all fields, ranging from improving comfort at your own house to controlling critical wide-range infrastructures:

Smart homes. By connecting household appliances, temperature sensors and security cameras to the Internet, a house owner could control his living comfort from any device at any time [2]. Instead of having to deal with many different switches and remotes around his house, he could have a single dashboard to monitor and control all of his devices. Such a dashboard could also allow for combined actions with multiple devices, for example activating the air conditioning also automatically closes the window blinds and the room doors.

Smart energy grids. With the shift from fossil resources to renewable resources, energy production is becoming more volatile and less predictable. Whereas coal and nuclear power plants can produce electricity at any time, the energy output of solar and wind plants depends on the time and weather conditions. This can lead to shortages or blackouts during calm cloudy days, or oversupply on windy sunny days. To deal with these fluctuations, smart energy grids

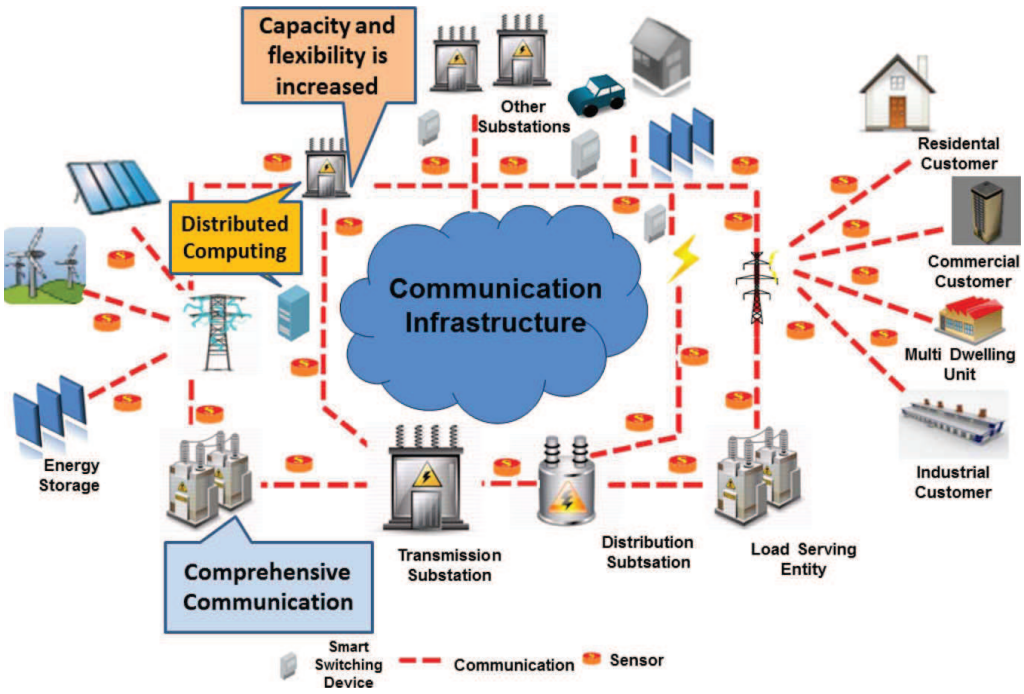


Figure 1.2: A smart energy grid architecture. [13]

should have the capability of predicting consumption behaviour on a more fine-grained basis, and shifting consumption peaks to better match production capacity [14]. For example, houses could be fitted with smart meters which continuously measure electricity consumption and report frequently to the energy producers so they can make better predictions. Smart washing machines could also be programmed to activate when the energy demand is low, in order to shift consumption peaks. These meters and appliances need to communicate with each other and with the rest of the energy grid, posing many information technology (IT) challenges for the supporting IT architectures and technologies [13], shown in figure 1.2.

Earthquake and volcano monitoring. To protect people living in seismically active regions such as fault lines and volcanoes, scientists must be able to predict future earthquakes or eruptions so people can be evacuated in time. Improving these predictions requires more frequent and accurate seismic measurements. However, sending humans into the field to measure seismic activity is dangerous. Instead, a network of monitoring devices can be deployed to collect measurements and report back to a base station [17] (illustrated in figure 1.3). These devices can make more frequent measurements, and can be deployed in more dangerous places where it is unsafe for humans (e.g. inside the crater of a volcano).

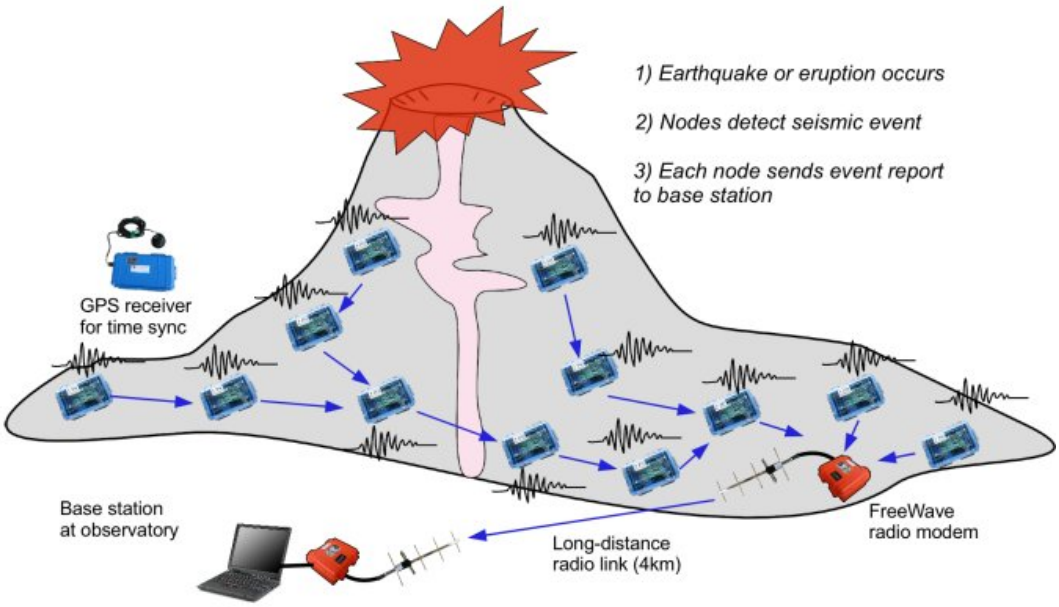


Figure 1.3: An IoT deployed by the Harvard Sensor Networks Lab on an active volcano to monitor seismic activity. [17]

1.1.2 Wireless sensor networks

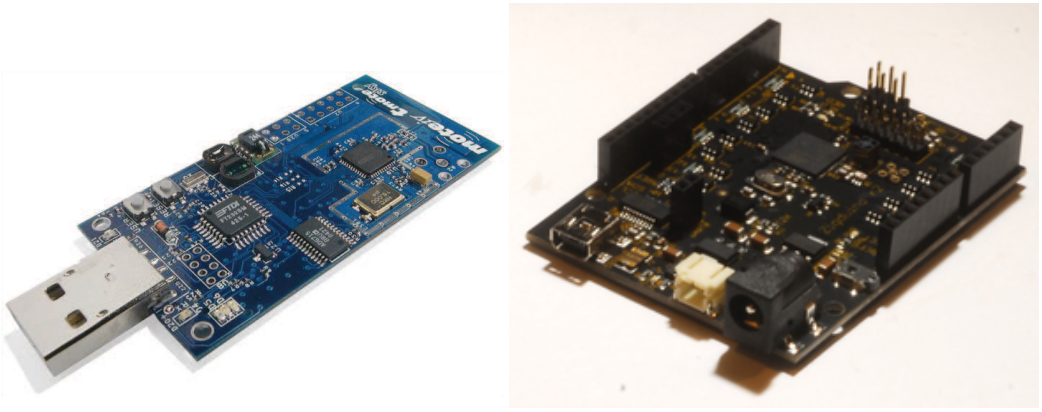
Many IoT applications are implemented using a wireless sensor network (WSN). A WSN consists of a number of small devices (often called 'nodes' or 'motes') equipped with sensors, actuators and a wireless antenna. These nodes measure changes in their environment using their sensors, communicate with each other and the outside world using their wireless connectivity and act upon their environment using their actuators.

These nodes have very different characteristics compared to regular computers, and have different design concerns for their programs:

- Due to their small size, they have extreme resource constraints. Whereas ordinary computers have multi-core GHz processors with many GBs of RAM, WSN nodes have single-core MHz microprocessors with only a few KBs of RAM. For example, the tiny TMote Sky has a 8 MHz microprocessor with 48 KB of random access memory (RAM) [5], while the larger AVR Zigduino r2 platform runs at 16 MHz with 128 KB of flash RAM [7].

- When deployed in remote outside locations (e.g. on a volcano or along a river), they have to be battery-powered or self-provisioning (e.g. small solar panel). This makes energy consumption a primary design concern for programmers, as the node's lifetime is constrained by its battery.

- Nodes have no infrastructural support: there are no routers or wireless access points to support their communication. All nodes are responsible for sensing



(a) The TMote Sky platform. (b) The AVR Zigduino r2 platform.

Figure 1.4: Sample platforms for WSN motes.

their environment, processing their data and routing messages across the network.

- WSN deployments are inherently unreliable. Nodes may fail due to their batteries running out or being destroyed by the environment (e.g. floods, fires or earthquakes). Transmitted messages may be lost due to bad weather conditions. Programmers must design with these failures in mind.
- WSNs are highly distributed. A network may consist of hundreds or thousands of nodes, with nodes moving in and out of range of each other all the time. Distributed programs must be able to scale up to large number of nodes and remain efficient.
- A single WSN may consist of many different types of devices with varying capabilities, specifications and architectures. Programs need to be compiled and configured to work on different device types.

In order to support wireless IPv6 connectivity for these low-powered devices, new Internet standards were introduced:

- IEEE 802.15.4 [1] specifies the physical network layer and media access control. It focuses on low power and low data rate wireless connectivity among inexpensive devices, as opposed to e.g. Wi-Fi which provides higher data rates but uses significantly more power.
- IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [10] allows IPv6 packets to be sent over IEEE 802.15.4 networks. The standard introduces various adaptations to achieve this, such as LoWPAN encapsulation and header compression.

To allow access from outside networks, a border router is usually deployed at the edge of the network. The border router translates between IPv6 over Low power

Wireless Personal Area Networks (6LoWPAN) packets and standard IPv6 packets and routes packets between the WSN and the rest of the Internet.

1.1.3 Peer-to-peer architectures

Peer-to-peer (P2P) architectures consist of peers with equal capabilities and responsibilities where the work load of the application is distributed over every peer. Peers communicate with each other to retrieve information needed to perform their part of the work, and provide information to other peers to help them with their work. Although peers may different in their resource capacity and processing power, they have the same role in the architecture and can take on the same kinds of tasks.

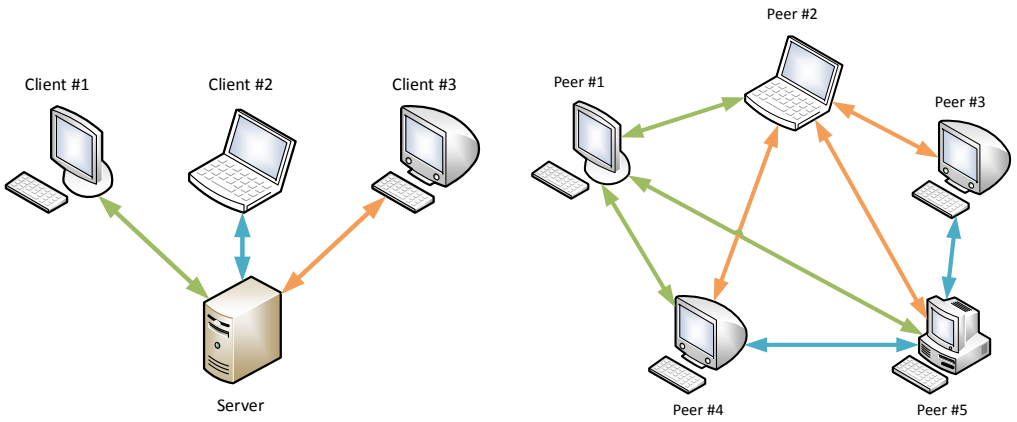
Peer-to-peer (P2P) architectures differ from the more classical client-server architectures, in which a server *provides* resources or services to be *consumed* by one or more requesting clients. Clients and servers are two distinct roles in the architecture with different capabilities responsibilities, and take up distinct parts of the application's work load. For example, the client is not concerned with how the server manages its resources and services, and the server is not concerned with what the client does with the provided resource or service.

Figure 1.5 gives an overview of client-server and P2P architectures. Figure 1.5a shows multiple clients requesting services from a single server, and figure 1.5b shows many peers communicating to request and provide services among each other. Note that clients in the client-server architecture always communicate with the server and do not communicate directly with each other, whereas peers in the P2P architecture can communicate with any other peer.

Many Internet applications use a client-server architecture. For example, the Web consists of web servers providing web pages and web browsers requesting them. Web browsers act as clients and are responsible for sending HTTP requests to the appropriate web server and rendering their responses as web pages on the user's screen. Web servers listen for HTTP requests and are responsible for producing a response. The web server can do all sorts of processing behind the scenes (e.g. accessing a database) to generate this response, but the client doesn't need to know about any of this and is only concerned with the server's response.

Peer-to-peer applications are less common, but have interesting applications. BitTorrent is a popular P2P file sharing protocol, where peers download and upload parts of a file to each other to help distribute the file to all peers (see 2.1.1). Cryptocurrencies such as Bitcoin [1] are a fairly recent invention, where peers verify and propagate transactions of a digital currency to ensure that every peer agrees on how much money every user has – without the need for a centralized authority such as a central bank. In both applications, every peer has the same role in the architecture and can communicate with any other peer.

[1] https://bitcoin.org/



(a) Client-server architecture.

(b) Peer-to-peer architecture.

Figure 1.5: Overview of server-client and P2P architectures.

1.2 Problem description

In order to keep wireless sensor networks operating for long periods of time, they must be able to adapt and evolve even after deployment. Over time, nodes may need to be updated with an updated program binary, a new configuration file or a new data set to be processed. However, it is often not practical and sometimes even impossible to collect a deployed node, take it back to a lab to be reprogrammed with the updated data and then re-deploy it in its original location. Therefore, these updates need to be done 'over-the-air': they must be distributed over the WSN and received by all destined nodes so these nodes can reconfigure themselves.

For such over-the-air reconfigurations, WSN nodes need to be equipped with the proper reconfiguration mechanisms. For example, in the current version (2.0.3) of the LooCI [15] platform for Contiki [6] nodes, the reconfiguration engine is responsible for listening for reconfiguration events, downloading the new program file over TCP and reprogramming the node with the downloaded program. This allows LooCI nodes to be completely reprogrammed remotely, without physical intervention. However, this solution is client-server based, with all nodes requesting the download from a single server. When the number of nodes becomes larger, this can put a lot of strain on both the server and the network as many TCP connections try to reach this single server. In contrast, a P2P approach would balance the load over all the nodes in the network by having them all participate in the program file's distribution. This allows it to be more battery-efficient on the nodes closest to the file's source, and more scalable when the network expands.

This master's thesis focuses on the file distribution mechanism for over-the-air reconfigurations. The objective is to design, implement and evaluate a file distribution protocol optimized for WSNs. This protocol could then be used in a complete reconfiguration solution, for example it could replace the TCP download in

LooCI, allowing for a more efficient program file distribution.

The peculiarities of wireless sensor networks (WSNs) (discussed in 1.1.2) pose many challenges for the design of a file distribution mechanism:

- File distribution requires large pieces of data to be transmitted in a short time, much larger than usually sent during normal operation of the WSN. The transmissions sent between nodes during normal operation most of the time fits in a single packet. This data usually consists of a few measurements, a new average or a notification to the other node – which all take up only a couple of bytes. However, large files such as programs or configurations do not fit in a single communication packet and cause much more traffic than normal operations. Since most of a node's energy is spent on its antenna and radio, a file distribution mechanism for WSNs should be efficient with the amount of packets needed to transmit the whole file in order to save the node's battery.

- The file should be distributed fast enough to allow nodes to return to their normal operation quickly. While a new program or configuration is propagating through the network, normal operations are largely halted. When nodes cannot be used for their intended purpose for a prolonged period of time, they might miss important measurements or fail to respond to critical events in the environment.

- Nodes should be able to continue file distribution even when one or more nodes fail, either temporarily or indefinitely. Nodes may need to re-download (a part of) the file and/or use a different route, but a single failure must not take down the whole network. When a failed node comes back online, it should be able to resume the file distribution, preferably without having to restart the download from scratch.

- Integrity of the distributed file must be guaranteed at every node. The file must be delivered correctly, since any error could corrupt the distributed program or configuration file. Since WSNs are unreliable, file distribution mechanisms must be able to verify the received file and recover from transmission errors.

- The file distribution mechanism must be able to target any subset of nodes in a WSN. WSNs can be heterogeneous, consisting of different kinds of nodes running different tasks. For example, a WSN to measure weather conditions could consist of different nodes to measure temperature, humidity or wind speeds. These nodes have different sensors and run different programs, but can be part of the same network.

1.3 Contributions

This master's thesis presents NanoTorrent as a possible solution for file distribution in wireless sensor networks.

NanoTorrent takes a peer-to-peer approach to the problem, where nodes act as peers and download and upload pieces of the file to each other. The design is inspired by the BitTorrent protocol, which has proven to be very successful on the Internet scale. By adopting ideas from BitTorrent, the solution can provide fast distribution speeds to many nodes without the need of an individual high-speed file server. However, it includes many adaptations to take advantages of the unique properties of WSNs (discussed in 1.1.2) and deal with their challenges (mentioned in 1.2)

The main contribution of NanoTorrent is the exploration of a hybrid peer discovery mechanism. BitTorrent mainly uses a tracker for discovering peers, while existing WSN distribution protocols only discover directly connected neighbour nodes. NanoTorrent combines both approaches into one solution, where the two mechanisms are used in parallel to connect with both close-by local peers and distant remote peers. This allows NanoTorrent to function in more heterogeneous WSN deployments, where not all nodes run the same program or are located on different networks.

The evaluation of the protocol shows that NanoTorrent can provide fast file distribution in many different network configurations. The distribution speed scales well with the size of the network, although in its current form the amount of transmissions is still an issue. The hybrid peer discovery approach means that peers send a lot of messages to discover other peers, and can find distant peers which require messages to take multiple hops across the network to reach them. More research and experimentation is needed to reduce the needed communications.

1.4 Structure of the text

- Chapter 1 is an introduction to the problem context of IoT and the challenges posed by file distribution for WSNs. It outlines the contributions of this master's thesis.

- Chapter 2 discusses the researched related work regarding P2P protocols on the Internet and in WSNs.

- Chapter 3 introduces the first half of NanoTorrent's protocol design, with the hybrid peer discovery mechanism. It describes both the centralized approach with a tracker managing the swarm, and the localized approach using link-local multicast announcements.

- Chapter 4 gives the other half of the protocol design, focusing on the file distribution aspect. It discusses the different parts of the protocol, such as how peers connect with each other, how they exchange pieces and how it takes advantage of locality.

- Chapter 5 explains the technicalities of the create prototype implementation. It describes the system architecture and its components such as the tracker, the border router and the peers.

- Chapter 6 discusses the evaluation of the protocol design and the prototype implementation. Experiments were performed in a simulated environment to analyse the scalability of the protocol, and its operation in a heterogeneous network.
- Chapter 7 concludes the text with a summary of all chapters, the lessons learned in making this thesis and possible future work building on this thesis.

Chapter 2

Related work

This chapter discusses the related work researched while designing the NanoTorrent protocol. Section 2.1 elaborates on peer-to-peer protocols for the Internet like BitTorrent, whereas section 2.2 looks at protocols designed for WSNs. As an exploration for a potential use case, section 2.3 looks at software evolution in WSNs. Section 2.4 concludes with a critical discussion of the researched material.

2.1 Peer-to-peer distribution for the Internet

2.1.1 BitTorrent

BitTorrent [4] is a peer-to-peer file distribution protocol designed to distribute large files to many clients over the Internet. All clients downloading the file also help uploading the file to others, helping to share the load of distributing to many clients.

BitTorrent is used to distribute operating system images (e.g. Ubuntu[1]), video game updates, free music and movies. This makes it one of the most popular file sharing protocols, accounting for 25% of upstream traffic in North America and Europe in 2014 [22].

Terminology

Piece A fixed-size part of the distributed file, which can be independently distributed and verified by peers.

Torrent info The metadata describing the file to be distributed by a torrent. This consists of the file's size, the number of pieces that make up the whole file, the size of each piece and the SHA-1 hashes of all pieces. This allows a peer to verify the integrity of the entire distributed file: the peer can calculate the SHA-1 hash of each received piece and check if it matches the expected hash from the torrent info.

[1] http://www.ubuntu.com/download/alternative-downloads

Torrent info hash The SHA-1 hash of the torrent info. This hash acts as an identifier for the torrent, used in messages by trackers and peers to identify the torrent. [2]

Metainfo file A `.torrent` file used to bootstrap a BitTorrent client. This consists of the URL of the tracker to use for this torrent, and the torrent info to describe and verify the file's contents.

Tracker A server tracking the state of the swarm of one or more torrents, allowing clients to discover other peers in the swarm.

Peer A BitTorrent client participating in a torrent's distribution.

Swarm The peers of a torrent.

Seed(er) A peer which has finished downloading the entire distributed file.

Leecher A peer which are still missing parts of the distributed file.

Initial seed(er) A seed to bootstrap the torrent's distribution. This seed has not downloaded the file from other peers, but has been set up by the distributor with the entire file already available.

Usage

In order to share a file using BitTorrent, a distributor creates a torrent metainfo file containing details about the file to be distributed and the tracker to use. The distributor then publishes this (small) metainfo file through other channels (e.g. on a website) for users to find it. To begin distribution, the distributor sets up at least one initial seed to bootstrap the distribution.

To download a file using BitTorrent, a user retrieves the torrent metainfo file through some other channel (e.g. a distributor's website) and launches their BitTorrent client with this file. The client joins the swarm as a leecher and starts downloading pieces from and uploading pieces to other discovered peers in the swarm. When all pieces are downloaded, the client becomes a seed: it keeps uploading pieces to help other peers in the swarm.

Protocol operation

The BitTorrent protocol consists of a tracker protocol (over HTTP) and a peer wire protocol (usually over TCP). Figure 2.1 shows an overview of the architecture.

The client joins the swarm by sending an *announce request* to the announce URL mentioned in the torrent metainfo file. The tracker receives this request, stores the new client's connection details and responds with an *announce reply* containing a

[2] Using a SHA-1 hash as unique identifier can (at least theoretically) cause problems due to hash collisions. However, these are usually ignored in real-life applications because they are incredibly rare. A single tracker or peer would have to handle a very, very large amount of torrents before the chance of a collision becomes realistic.

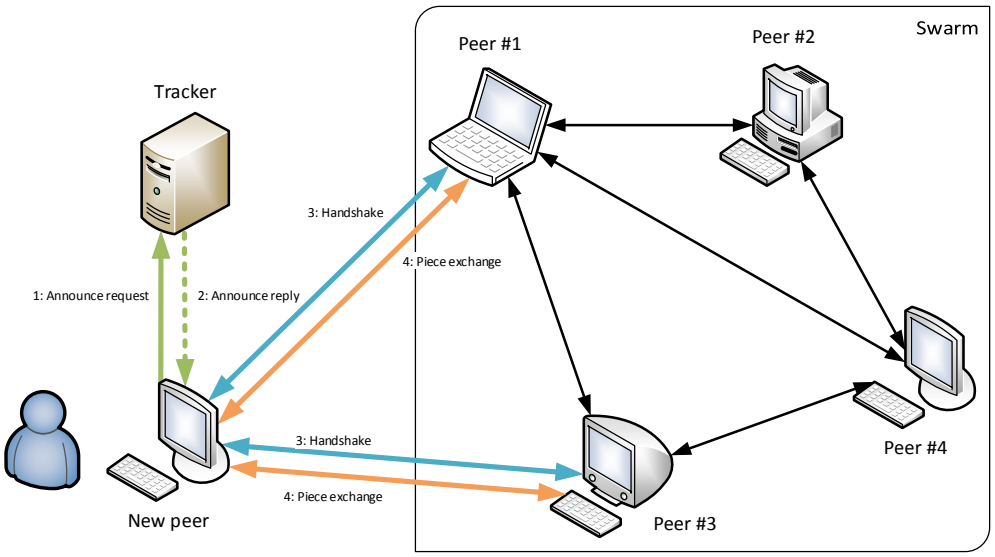


Figure 2.1: Overview of the BitTorrent architecture.

list of other peers with whom to connect. When other peers join the swarm, these stored connection details can be returned in their announce replies, inviting the new peers to connect with this client.

After retrieving a list of peers, the client opens TCP connections with these peers. They both send a handshake message to ensure that both ends are indeed operating on the BitTorrent protocol and are trying to download the same torrent (identified by the metainfo hash).

After the handshake, the peers can start sending peer wire messages over the connection. The `have` and `bitfield` messages are used to notify about which pieces this peer has completely downloaded, so the other peer can potentially request them. The `request`, `piece` and `cancel` messages are used to transfer data of a single piece. This set of protocol messages is already sufficient to implement the file transfer. When a client has fully downloaded a piece, it sends a `have` message to all other peers so they are made aware of the newly available piece and may later `request` it. This continues until the client has downloaded every piece, at which point the client becomes a seed: there's nothing left to download, but it keeps uploading to help other peers in the swarm.

However, BitTorrent must also deal with limited network conditions and potentially 'rogue' implementations of the protocol. TCP's congestion control performs poorly when sending over many connections at once, so not all opened connections should be used at the same time. Also, a 'selfish' client implementation could ignore all received `request` messages, and only download from others without helping others by uploading to them. Therefore, BitTorrent clients maintain extra flags for every peer: the *choke* flag means that the other end is not allowed to request data, and the

interested flag means that the other end would like to request data from this client. A peer sends `choke` or `unchoke` messages to notify whether this peer is choking the other end, and `interested` or `uninterested` messages to notify whether this peer is interested in one of the other's pieces. Clients can then use a choking algorithm to control which peers are allowed to download (by 'unchoking' them), and which peers should be (temporarily) blocked (by 'choking' them). For example, clients can keep only a limited number of connections unchoked, to prevent congestion. To promote 'good behaviour' for clients, most BitTorrent implementations uses a *tit-for-tat strategy* for their choking algorithm. With this strategy, peers will choke the other end when they are being choked (hence tit-for-tat), and will try to unchoke the other end when they are unchoked and the other end is interested.

Advantages and disadvantages

Every peer downloading missing pieces of the torrent's file also uploads the pieces it already has to other interested peers. This peer-to-peer file sharing approach has many advantages for both clients and distributors:

- Instead of having a single file server serving all clients wanting to download the file, BitTorrent employs the (often underused) upload capacity of those clients to help distributing the file to other clients. The combined upload capacity of these clients can be very large, and comes almost free to the distributor. The distributor only has to invest in a modest server acting as an initial seed, and letting the clients share the load.

- Clients can download many pieces from many peers in parallel, which can lead to faster downloads.

- BitTorrent is more failure-resilient than regular file distribution protocols such as FTP. The FTP server is a single point of failure: nobody can download the file if the server fails. In contrast, the failure of a single BitTorrent peer does not greatly impact the rest of the swarm. If every piece owned by the failed peer is still available at some other peer, all remaining peers can still download the full file. When the failed peer restarts, it also doesn't need to restart the complete download: it can verify the previously downloaded pieces using the checksums and resume downloading the missing pieces.

- Regular file distribution protocols do not scale well when the number of simultaneously connected clients increases. FTP requires 2 TCP connections for each connected client trying to download a file, which can quickly consume the server's resources. The server also uses more upload bandwidth, which can cause congestion in the network. The classic solution to this involves installing more servers ('mirrors') in more locations, but this is expensive. In contrast, the load on the initial seed in BitTorrent actually *decreases* as a torrent becomes more popular: all connected clients share the load to help others download the file.

However, there are also disadvantages to moving to peer-to-peer distribution:

- Unpopular torrents can become abandoned, with no active seeds in the swarm. This means that no new peers can download the full file, rendering the torrent useless until a seed is re-introduced. To prevent this, the distributor should maintain at least one seed in the torrent. This ensures that in the worst-case scenario, the whole file can be retrieved from the distributor's seed.

- Moving the upload load from the distributor to the clients means that the networks of clients see more upload traffic. internet service providers (ISPs) optimize their consumer networks for 'regular' consumer traffic behaviours, which consists primarily of download traffic. An ISP might throttle torrent traffic in order to 'shape' the traffic to a desired profile, reducing the performance of BitTorrent peers on its network.

2.1.2 Local Peer Discovery for BitTorrent

Local Peer Discovery is a BitTorrent protocol extension designed to support the local discovery of peers, aiming to maximize torrent traffic through the local area network (LAN) and reduce traffic through the ISP [26]. Although there is no formal specification published as of the time of writing [3], it is quite consistently implemented in various BitTorrent clients and supports both IPv4 and IPv6 multicast.

The extension introduces a new discovery mechanism parallel to the centralized tracker, which uses UDP messages periodically sent to a local multicast address to announce and discover peers in the local network. These messages are similar to standard announce requests, and include the torrent info hash and peer connection details. When other peers in the LAN receive this multicast message, they can read the connection details and try to establish a peer connection.

Local Peer Discovery is not used very often in practice, as home user LANs are small and it is fairly uncommon to have many users downloading the same torrent inside the same local network. It makes much more sense in WSNs where many peers do share the same local network, which is why this peer discovery mechanism is included in the protocol design (see 3.2).

2.2 Peer-to-peer distribution for WSNs

2.2.1 Deluge

Deluge [16] is a reliable data dissemination protocol for propagating large data objects from one or more source nodes to many other nodes in a WSN. It allows incremental updates of large files to be efficiently flooded to the whole network.

Deluge assigns a monotonically increasing *version number* to each new version of the distributed object, and subdivides the object into fixed-size pages. Since new versions often only have small difference compared to the previous version, many

[3] Due to this lack of published specification, the only decent source [26] is on Wikipedia.

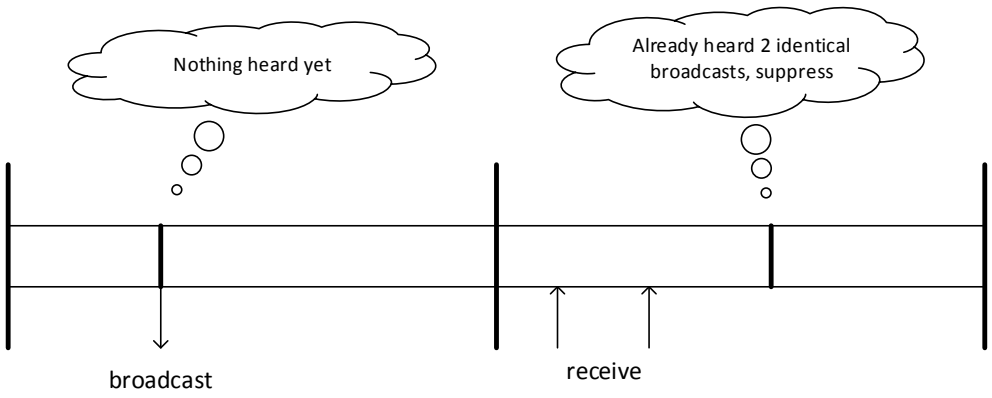


Figure 2.2: Example of Trickle's polite gossip algorithm. In the first interval, the node hasn't heard any other broadcasts yet and sends its broadcast. In the second interval, two identical broadcasts have already been received before the own broadcast is due, therefore the node suppresses its redundant broadcast.

pages do not change between versions. Deluge uses this information to re-use pages from previous versions and prevent redundant page transfers. Every new page is identified by a monotonically increasing *page number*.

Deluge nodes send periodic local broadcasts announcing their current version number and the latest available page (with the guarantee that all previous pages are also available). To control the transmission of potentially redundant announcements, Deluge uses the Trickle algorithm [18] to do a 'polite gossip'. For example, when a node hears a lot of its neighbours announcing the same version that it already has, it knows that there's no point in announcing its own version again. Most of its neighbours will probably already have heard one of the other neighbour's messages, so the node stays quiet to avoid broadcasting a redundant message.

Figure 2.2 shows an example of the Trickle algorithm in action. Before every interval, the node picks a random time $t \in [0, \tau]$ at which to schedule its own broadcast. If it doesn't hear a certain number of other identical broadcasts, it will send its broadcast at time t. If it does receive enough identical broadcasts, it will suppress its own broadcast at time t and will not send anything during this interval. Because t is randomly re-generated after every interval, the cost of sending broadcasts is distributed among all neighbours, ensuring that nodes use on average the same amount of energy for maintaining the announcements.

When a node hears one of its neighbours announcing a newer version number, it must update to the new version. Since many of its neighbours will hear this same announcement and likely also need to update, it would be wasteful for all these nodes to send requests to retrieve the new pages. Once again, Trickle is used to prevent redundant messages, so only a couple of nodes will send a broadcast to request the new page and only a few nodes will reply to the request. This allows new pages to

propagate very efficiently throughout the network.

Deluge takes great advantage of local broadcasts in WSNs, and makes flooding code updates very efficient. However, it assumes that all nodes in the network all need the same file, which might not be the case in heterogeneous deployments. NanoTorrent takes some advantage of local broadcasts by sending piece data as local multicasts (see 4.5), however it can also use unicast messages to reach distant peers in a heterogeneous network.

2.2.2 TinyTorrents

TinyTorrents [20] is a peer-to-peer protocol designed for data dissemination in WSNs. Like BitTorrent, TinyTorrents uses torrent files to describe the tracker's location and the file's contents. Peers join the swarm by contacting the tracker, and discover other peers with whom to connect and exchange data.

TinyTorrents recognizes that many of the problems BitTorrent tries to solve on the Internet also match the challenges posed by WSNs. BitTorrent balances traffic load with its P2P approach and avoids network partitions with a tracker. Data integrity is ensured with piece-wise checksums, and data replication is made efficient with its rarest-first piece selection policy. TinyTorrents adopts these design decisions, and similarly NanoTorrent follows the same reasoning.

However, BitTorrent's peer wire protocol consists of many short frequent messages such as `have`, `choke` and `interested` (explained in 2.1.1). These are decremental for WSN nodes, as they consume a lot of energy for transmission. TinyTorrents solves this by combining separate `have` messages into one less frequently sent `bitfield` message, and using a static approach based on the peer's position in the swarm to fairly distribute the load among seeders and leechers. NanoTorrent also drops BitTorrent's short `have` messages, but its tracker does not currently take fairness into account.

TinyTorrents supports interoperability with standard BitTorrent through a gateway acting as a bridge between the two protocols. The gateway can convert TinyTorrent torrents into BitTorrent torrents, allowing regular BitTorrent clients to download data from inside the WSN and allowing WSN nodes to download torrents published outside the WSN. This allows remote users to access accumulated sensor measurements from the WSN, or to reprogram nodes from anywhere on the Internet. NanoTorrent does not currently interoperate with BitTorrent, but can support interoperation with nodes on other WSNs or the Internet thanks to its IPv6 foundation.

The authors of TinyTorrents recognize that their reliance on a centralized tracker is a weak point of their protocol, which is a single point of failure in their system. NanoTracker attempts to alleviate this by combining multiple peer discovery mechanisms, so peers can still discover each other even if the tracker (temporarily) becomes unavailable.

2.3 Software evolution in WSNs

2.3.1 Challenges and approaches for reprogramming WSNs

Wang et al. [25] outline a framework to examine the different functions for a reprogramming solution for WSNs, and survey the approaches taken by existing solutions.

Figure 2.3 shows their framework for reprogramming solutions. This includes features such as version control to manage different software versions, scope selection to control which nodes are affected by a reprogramming operation and code acquisition to request the download of a new version.

The authors note that WSNs come with many challenges. The algorithms used on a WSN node must be well designed to fit the constrained capacity profile of the node, they must be scalable and energy-efficient, and they must be able to handle unreliable communications.

The paper groups the approaches of various existing code dissemination protocols like Deluge by categories, such as the scope of an individual reprogramming operation, the ability to traverse a multi-hop network and the use of pipelining to accelerate the distribution. Although many protocols support multi-hop reprogramming, only a few such as Aqueduct and TinyCubus have support for a limited scope reprogramming.

Important objectives for a good reprogramming solution are reliability, scope coverage and autonomy: all targeted nodes must be able to autonomously and correctly retrieve the new software version. For practical usefulness, it must also be fast enough to cause minimal disruption to the normal operation of the WSN, consume only a minimal amount of energy during the transmission and use only a minimal amount of RAM.

In their framework, a P2P file distribution protocol for WSNs could fulfil the code acquisition and code dissemination functions of a network reprogramming solution. When a peer finds out that it should update to a new version, it can connect with the other peers with that same version and start exchanging the file with them.

2.4 Critical discussion

BitTorrent has shown that P2P file distribution can be made fast and easy-to-use on the Internet. It introduced a lot of good ideas, such as the choke/interested flags to control network load and punish misbehaving peers with a tit-for-that algorithm. However, these short frequent control messages can cause a lot of network traffic, which is the main cost factor for WSN nodes. The design of a protocol aimed at WSN nodes should be more careful with its transmissions.

BitTorrent Local Peer Discovery shows that multicast messages can be used to discover other peers on the same LAN for faster local file distribution. However, multicast is not well supported in the current Internet infrastructures, many of which still use IPv4. It is also very unlikely to have multiple BitTorrent clients on the same network downloading the same torrent, as consumer LANs are usually very small and corporate LANs mostly refrain from allowing BitTorrent traffic on their

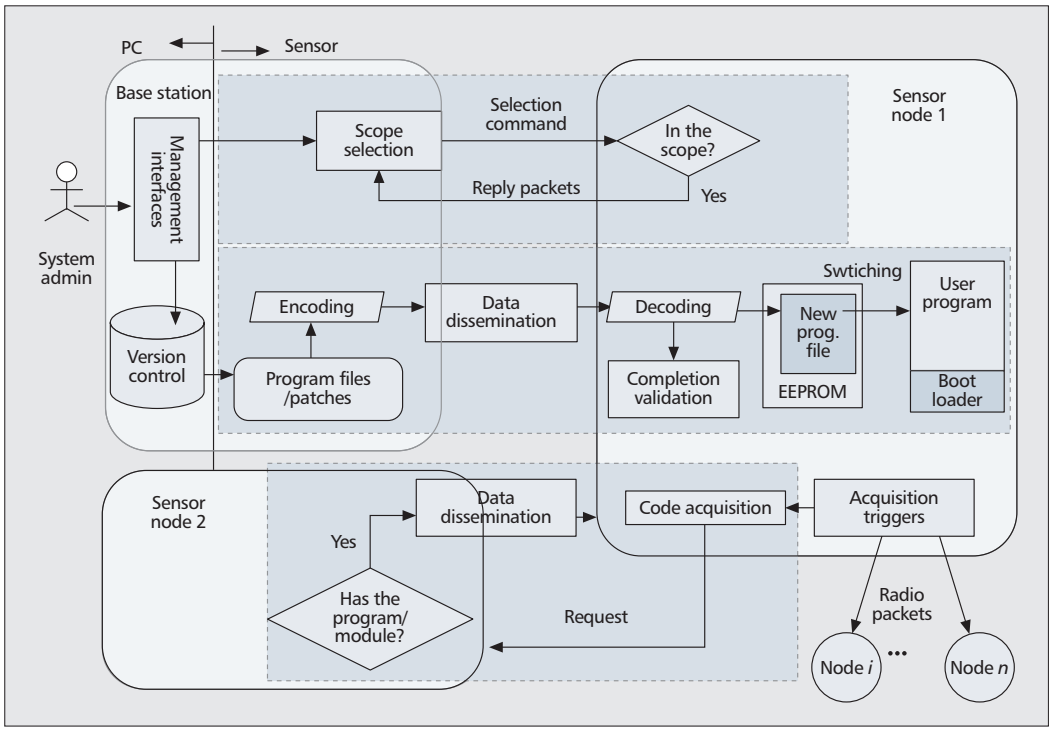


Figure 2.3: Framework for WSN reprogramming solutions. [25]

networks. This makes BitTorrent's Local Peer Discovery very underused on the Internet. However, the ideas behind its design are sound, and can be useful in the context of WSNs.

There exist already a number of P2P file transfer protocols for WSNs. Deluge uses local broadcasts and listens for other neighbour's messages to prevent redundant transmissions. TinyTorrents is also influenced by BitTorrent's design, and even provides interopability with the original prototocol through the use of a gateway. However, none of these protocols leverage the IPv6 architecture which is becoming more common in WSNs deployments as base for their communication. Deluge uses lower-level broadcasts for its communications, while TinyTorrents uses a custom TinyHop routing protocol. This opens an opportunity for a file distribution protocol which can run on top of the IPv6 network stack, allowing multi-hop connectivity between distant nodes in the network and even communication with outside networks.

A P2P file distribution protocol for WSNs could fulfil the code acquisition and code dissemination functions of a network reprogramming solution in the framework of Wang et al. [25]. Their paper also investigates protocols such as Trickle and Deluge, and notes that they only allow for full network reprogramming. There is an opportunity for a file distribution protocol that allows reprogramming of a limited scope of nodes, possibly with multiple reprogramming operations operating in parallel targeting different groups of nodes.

Chapter 3

Peer discovery

This chapter discusses the design of the peer discovery mechanisms in NanoTorrent. These mechanisms enable peers to discover other peers downloading the same torrent, allowing them to connect with each other and exchange file data. NanoTorrent uses a hybrid approach consisting of two peer discovery mechanisms running side-by-side, allowing it to exploit locality while at the same time keeping distant nodes connected.

Section 3.1 describes the client-server approach using a centralized tracker. Section 3.2 discusses the local peer discovery mechanism using IPv6 local multicasts. Finally, section 3.3 concludes with a discussion of the hybrid approach.

3.1 Peer discovery with centralized tracker

One approach to implementing peer discovery is using a client-server architecture where membership information is provided by a central server known as the *tracker*. This tracker maintains the state of all peers in the swarm, and peers can discover and be discovered by other peers by sending requests to obtain (parts of) this information.

3.1.1 Requirements

The tracker is responsible for maintaining the *swarm* of each of its torrents, i.e. the list of peers for every tracked torrent. It must be notified when a peer wants to join the swarm to start downloading a torrent, and when it wants to leave after it finishes downloading and seeding the torrent. In order to deal with possible peer failures, messages from peers to the tracker are also treated as a 'heartbeat' for that peer. The tracker keeps track of the timestamp of the last message from each peer and periodically removes peers who have not contacted the tracker for a long time. This helps to ensure that peers will discover active peers most of the time and will not try to connect with unreachable stale peers.

Peers should be able to discover other peers by contacting the tracker and receiving a list of other peers with whom to connect. Each peer is responsible for keeping the tracker up-to-date on its own participation state, such as when it joins or leaves the swarm. This allows the peer to be discovered by other peers when they

contact the tracker themselves. In order to keep their heartbeat alive at the tracker, peers need to periodically contact the tracker to refresh their state as long as they're participating in the swarm.

3.1.2 Torrent descriptor

Before a file can be distributed to a set of nodes in the network, these nodes must have enough information to download and verify the whole file on their own. This information is contained in a *torrent descriptor* which is generated upfront and needs to be made available at every node to start the file transfer.

The torrent descriptor is similar to the *metainfo* in BitTorrent, but uses a denser structure than BitTorrent. This allows for a shorter descriptor file compared to BitTorrent's `.torrent` file, and faster parsing by nodes.

A torrent descriptor consists of the IP address and port of the tracker, and the torrent information (further explained in 4.2. Like BitTorrent, trackers and nodes use the torrent info hash calculated from the torrent information as an identifier for a torrent (see 2.1.1). This way, a node only needs the torrent descriptor to locate the tracker and join the swarm corresponding to the correct torrent.

The distributor must generate the torrent descriptor and make it available to all destined nodes before starting file distribution. How the torrent descriptor is made available at a node, is out of the scope of the protocol.

3.1.3 Protocol

Peers communicate with the tracker through a request-reply protocol, (unimaginatively) named NanoTracker. This is a simple request/reply protocol over UDP, where peers send an *announce request* to the tracker, and the tracker responds with an *announce reply*. Since the tracker can potentially track multiple swarms for multiple torrents, every message also includes the info hash of the torrent to identify the swarm. In theory, this allows a peer to join multiple swarms and download multiple torrents concurrently, although the current implementation supports only one concurrent torrent download.

The address of the tracker is included in the torrent descriptor, which must be available to every peer to participate in the torrent. A peer sends an announce request to this address over UDP, including the event of the request and the maximum number of peers that it wants to receive in the reply. Since different clients may have different resource constraints, a client may want to request more peers from the tracker if it wants to connect with more peers simultaneously. The event indicates the reason for sending the request, and determines how the tracker should react to the request:

started The peer wants to start downloading the torrent. The tracker should add the peer to the swarm, and allow it to be discovered by other peers.

stopped The peer has stopped downloading the torrent. The tracker should remove the peer to the swarm, so it will no longer be discoverable by other peers.

refresh The peer wants to refresh its participation in the swarm. The tracker should update the peer's heartbeat timestamp, ensuring that it stays discoverable for the near future.

completed The peer has finished downloading the torrent and is now seeding the torrent. The tracker could use this information for statistics, or for prioritizing the peer in the replies for other peers.

The tracker selects a number of other peers from the swarm, making sure it does not exceed the maximum number of peers requested by the peer. The tracker then constructs and sends back an announce reply, which contains a list of IPv6 addresses of the selected peers. The peer stores these addresses and can then start connecting and exchanging pieces with those peers as described in chapter 4.

3.1.4 Comparison with BitTorrent

The design of the tracker protocol is heavily inspired by the UDP Tracker protocol for BitTorrent [23], but is adapted to the needs of WSNs.

IP spoofing prevention

BitTorrent tries to prevent peers from 'spoofing' their IP addresses, i.e. using a different address as the UDP source address than their own. A malicious user could abuse the peers in a very popular torrent's swarm for a distributed denial of service attack (DDoS) by spoofing an announce request with the victim's IP address. Since the BitTorrent tracker stores the source IP address of the announce request and announces it to other peers looking to discover peers in the swarm, these peers can be tricked into sending traffic to the uninterested victim.

To prevent IP spoofing, a peer has to first send a connect request and receive a connection identifier in the tracker's connect reply. This identifier must then be passed in every announce request and validated by the tracker. This way, the peer has to 'prove' it is using its own IP address by presenting information sent to the provided source address.

In the context of WSNs, IP spoofing is not really a concern. Therefore, there is no need for an initial connect request/reply step and peers can immediately send announce requests.

Random peer ports

In BitTorrent, peers choose a random TCP port to listen for incoming connections from other peers for the peer-to-peer protocol [4]. This allows multiple torrent downloads to run on different ports without interfering with each other. However, this also means that other peers need to know both the IP address as well as the port number of a peer in order to set up a connection. Therefore, a peer includes its port number in the announce request, the tracker stores this port along with the rest of the peer's information and announce replies contain both the IP address and the port number for each peer.

For WSNs, it is more convenient if all peers use a fixed port number for peer-to-peer communication. This allows local peer discovery (detailed in 3.2) to use a single multicast address and a single port to reach all local peers. A disadvantage of this approach is that now all peer messages must include the torrent's info hash to identify the torrent, as the same port could be used by multiple torrents being downloaded by different nodes in the network.

Download and upload statistics

In BitTorrent, peers must provide information about the amount of bytes they have uploaded and downloaded, as well as how much data they still have left to download in order to complete the torrent download. This statistical information can be used by the tracker for example to implement more sophisticated peer selection algorithms for generating announce replies, to analyse the 'health' of the torrent or simply to display on the tracker's website.

In the context of WSNs, this information is not really needed and therefore not supported by the NanoTracker protocol.

3.1.5 Conclusion

By centralizing membership information at a single tracker, it becomes easier to analyse the state of the swarm. An operator of the WSN can query the tracker at any time and retrieve information about which peers are currently connected. The tracker can detect when a peer stops sending periodic announce refreshes, and notify the operator of a potentially failed peer.

However, the need to periodically refresh the swarm membership state of every peer inevitably leads to additional network traffic. It could also create a bottleneck in the section of the network closest to the tracker, where all messages from all peers in the WSN need to pass.

3.2 Peer discovery with local multicast

In many WSNs, nodes with similar tasks are often deployed closely together. Some networks consist of small clusters of similar nodes, or only consist of one node type altogether. In these deployments, peers need the same files as their immediate local neighbours, and can distribute files more efficiently through local broadcasts to all of their neighbours (see 4.5).

To encourage peers exchanging pieces of the distributed file with their immediate neighbours, peers need to be able to discover neighbour nodes which are downloading the same torrent. The local peer discovery mechanism achieves this by periodically sending an announcement as a multicast to all immediate neighbours. If a peer receives such a local advertisement, it can try to connect with the originating peer.

3.2.1 IPv6 link-local multicast

IPv6 supports three addressing modes: unicast, anycast and multicast [8]. A unicast address identifies a single IPv6 interface, and a packet sent to it is delivered only to that interface. An anycast address identifies one or more interfaces, but a packet sent to it is delivered to *one* of those interfaces. A multicast address also identifies multiple interfaces, but packets are delivered to *all* of those interfaces. This is a big improvement over IPv4, which only defines unicast and broadcast addresses (with *optional* multicast support).

Orthogonal to addressing modes are address scopes, which indicate in which part of the network a given address is valid. Global addresses are valid over the whole Internet, whereas link-local addresses are only valid in a single network.

IPv6 reserves some special multicast addresses to send messages to certain types of nodes in an address scope. For example, a multicast address can target all nodes, all routers or all DHCP servers on the same link, network or organization.

3.2.2 Protocol

The local peer discovery protocol consists of periodically sending an announcement to all direct neighbours. This allows local peers listening to these announcements to discover new peers and in return set up connections with them.

For the purpose of discovering other peers in the immediate neighbourhood, NanoTorrent currently uses the `ff02::1` link-local all-nodes multicast address. This address works like a link-layer local broadcast address (e.g. the MAC address `ff:ff:ff:ff:ff:ff`). All nodes already listen to incoming packets on this address, requiring no extra configuration.

Rather than adding another special-purpose message type for periodic announcements, NanoTorrent re-uses the `have` message which is already periodically exchanged between peers (discussed in more detail in 4.3). Peers use this when setting up a connection to a newly discovered peer, and to periodically inform connected peers of their download progress. Local peer discovery is implemented by also sending these periodic `have` messages to the link-local all-nodes multicast address. When a local peer receives any message from an unknown peer, it will try to set up a peer connection and (eventually) send back their own message. This reply allows the originating peer to set up its own connection with the discovered peer, resulting in a two-way connection where both peers can request pieces.

Figure 3.1 shows a sequence diagram of a possible interaction between three neighbouring peers. The first peer sends its periodic announcement as a multicasted `have` message to all its neighbours. When they receive this message, they can choose to set up a connection with the peer if they have enough resources to do so and are downloading the same torrent. In this case, the third peer detects that the new peer has interesting pieces, and immediately sends a data request for a missing piece. The first peer receives this message from an unknown peer, which allows it to discover the third peer and add it to its own connections. It then dutifully responds to the request, and can start requesting pieces for itself. If the first peer did not have any

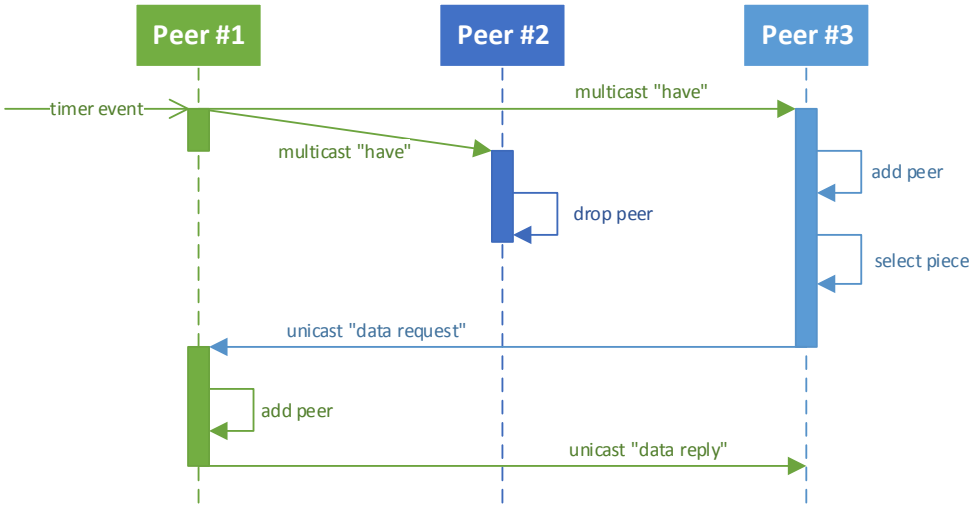


Figure 3.1: Sequence diagram of local peer discovery. The second peer has no more resources to add a connection for the first peer, and ignores the announcement. The third peer does add the new peer, and immediately requests a piece from the new peer. This allows the first peer to discover the third peer, add it and reply to the request.

pieces that are immediately interesting to the third peer, the third peer will later on send its own multicast announcement, so the first peer can eventually discover the third peer.

3.2.3 Conclusion

Local peer discovery can be added to NanoTorrent with minimal additional cost. For the wireless radio antenna, it doesn't matter whether a packet is destined to one or multiple nodes: all messages must be broadcasted anyway over the wireless medium. However, multicast packets do take more time for nodes to process, as they cannot short-circuit them like unicast messages. When a node receives a unicast packet destined to a different node, it can reject this as early as the link layer (using the destination MAC address) or the network layer (using the destination IP address). All-nodes multicast messages however always make it through to at least the transport layer, only rejecting the packet when the node is not acting as a NanoTorrent peer and therefore is not listening on the protocol's UDP port.

By building on top of IPv6 multicast, local peer discovery can work on any IPv6-compliant networking stack. Rather than going to a lower layer protocol such as IPv6 Routing Protocol for LLNs (RPL) [11] for information about physical neighbours, NanoTorrent stays generic and independent of the underlying physical network

infrastructure. This also allows it to re-use large parts of the existing protocol which is already designed for IPv6.

3.3 Conclusion

Like in BitTorrent, the centralized tracker acts as a single source of (partial) truth about all active peers in a torrent's swarm. Every peer periodically announces itself to the tracker to keep its membership state alive, and in return received a list of other known peers in the swarm. Since this list is a random subset of the whole swarm, the chance of the P2P network becoming partitioned is fairly low.

Local peer discovery ties in nicely with the peer wire protocol for the actual file distribution (see 4.3), and allows for a hybrid peer discovery solution. The tracker is still useful for ensuring that peers don't end up isolating themselves by only connecting to local peers, while local peers can efficiently propagate newly retrieved pieces to all of their immediate neighbours at once (see 4.5).

Chapter 4

Peer-to-peer file distribution

This chapter discusses the design of the file distribution mechanism in NanoTorrent. These mechanisms enable peers to connect with each other, exchange download progress information and request file data.

Section 4.1 lists the requirements for the distribution. Section 4.2 explains the use of fixed-size pieces to subdivide the distributed file. The management of peer connections is discussed in 4.3, and the actual piece exchange is described in 4.4. The local peer discovery mechanism described in 3.2 allows for an optimization in the piece exchange, which is discussed in 4.5. Section 4.6 rounds up the chapter with a conclusion.

4.1 Requirements

The distributed file must fit in the non-volatile storage of every targeted node, and thus its size is limited. Whereas files distributed on the Internet can be several GBs in size, WSN nodes can only store files in the orders of a couple of KBs. For example, the electrically erasable programmable read-only memory (EEPROM) of the AVR Zigduino r2 platform, (used in the evaluation in chapter 6) can hold up to 4 KB of data [7]. The file distribution protocol is dimensioned to work well with these relatively small files, in its current form up to 64 KB (2^{16} bytes).

The protocol should attempt to prevent redundant communications. For examples, peers should not request parts of the files from another peer if they do not know whether the other peer already has that part available. Therefore, peers should advertise information about their own download progress to others so they can make informed requests. This implies that peers need to retain stateful information about other peers.

Peers should try to balance the overall load on the P2P network. If a certain piece of the file were only available at one single peer (which is often the case with an initial seed early on in the swarm's lifetime), that peer could become overloaded when other peers do not properly coordinate their requests. In the worst case, a failure of that single peer could prevent the rest of the network from receiving the

whole file. The protocol should try to relieve peers with highly wanted file pieces, and prioritize their distribution over more commonly available pieces.

File distribution must be robust against corruptions and transmission errors. For example, when distributing a new program binary, it is crucial for the program's correctness and stability that the received file is bit-for-bit identical to the original. Therefore, peers must be able to independently verify the integrity of the received file in its entirety.

4.2 Pieces

NanoTorrent uses the same approach as BitTorrent to subdivide the distributed file in fixed-size *pieces* which can be independently distributed over the P2P network. This allows peers to retrieve them in any order and download multiple pieces from multiple peers in parallel (see 2.1.1).

All information about a torrent's pieces are contained in the *torrent information*, which is part of the torrent descriptor (detailed in 3.1.2). Peers can verify the integrity of the entire file by verifying the integrity of each individual piece. The SHA-1 hash of the torrent information (further abbreviated to *info hash*) is used to tag all messages exchanged between peers, to ensure that messages for different torrents do not interfere with each other.

4.3 Peer connections

Peers can discover other peers in two ways: they can find out about another peer through the tracker (detailed in 3.1), or they receive a message from another peer (which includes local peer discovery from 3.2). Both give the peer an IPv6 address of another peer, which they can use to directly communicate with the peer.

However, peers need to maintain some connection state for each peer they are communicating with:

- The peer's IPv6 address, for sending messages. This also acts as a 'key' for the connection's state, allowing the peer to find the correct connection state when receiving messages from the peer.
- The set of pieces available at that peer. This allows the local peer to find out which missing pieces are available at which connected peers, so it can send piece requests to the right peer.
- The time of the last message received from that peer. This acts as a heartbeat, so the local peer can clean up this connection state if the other peer stops sending periodic messages.
- Information about the current pending piece request to that peer. This allows the peer to track which pieces it is expecting to receive from which peers, and what parts of the pending pieces it has already received (see 4.4).

The protocol specifies two types of messages to manage peer connections: `have` and `close` messages. A `have` message informs others of a peer's set of currently available pieces. Peers send this message periodically to keep their heartbeat alive at other peers, and when they have made new progress on their own download and want to notify others. A `close` message informs another peer that it should close the connection with the sending peer and clean up its connection state. This message is sent when a peer decides to close one of its opened connections.

To set up a connection, a peer simply allocates some connection state locally and can then start sending messages to the other peer. Depending on how the other peer was discovered, the first message sent could be a periodic `have` message (when the peer doesn't know anything yet about the available pieces), or an immediate piece data request (when the peer already knows about some of the available pieces).

To maintain its peer connections, a peer must periodically send a heartbeat to inform them about its own set of available pieces (using a `have` message). This lets other peers that this peer is still alive (and its heartbeat timer should be refreshed) and updates them on the current set of available pieces. If the other peer does not yet have a connection state for this peer, it can decide to set one up so it can send requests to this peer.

Keeping the view on the available pieces at other peers up-to-date is critical to achieve a fast file distribution to all peers. Therefore, this information is also piggy-backed onto every exchanged peer message. Peers could also decide to send their next *have* heartbeat immediately after completing the download of a piece. However, this has the risk of flooding the network when many pieces are completed at around the same time. As a compromise, the next heartbeat only gets re-scheduled to a slightly earlier time whenever a piece is completed.

Since peers need resources to allocate and maintain their peer connections, the amount of simultaneously open peer connections is limited. This also means that when one peer creates a connection to another peer, the other peer doesn't necessarily have to do the same. In that case, the other peer *immediately* sends a `close` message to indicate that it won't be able to maintain its side of the connection and the original peer should clean up its connection state. After cleaning up the failed connection, the peer can continue to try connecting with a different peer instead.

One exception to these fast `close` semantics shows up when taking into account the multicast `have` messages used in local peer discovery (see 3.2). Whereas regular `have` messages are sent to initiate a connection to a *known* peer, these multicast messages are sent to discover one or more *unknown* neighbouring peers. If every peer receiving a multicast advertisement would reply with a useless `close` message, the protocol's performance would degrade drastically. To prevent this, peers only send a `close` reply when the received `have` message is directed to only them (i.e. when the message is destined to a *unicast* address as opposed to a multicast address).

4.4 Piece exchange

The protocol provides request-reply style messages to exchange piece data with `data request` and `data reply` messages. A `data request` includes the index of the requested piece (following the order in the torrent information) and the offset of the first data byte in the response. A `data reply` contains the same fields, followed by the actual data bytes. Peers can freely choose how many bytes they send in a `data reply` depending on their own capabilities, although they should try to fill the whole packet with piece data to reduce the total number of requests needed to exchange the whole piece.

A peer starts by selecting a piece to request. They can use any piece selection strategy, however the current implementation uses the same *rarest-first* policy as BitTorrent. This policy selects the piece which is the *least common* of all pieces available at all currently connected peers, in an attempt to help make these 'rare' pieces more common in the network and balance the overall load.

Following, the peer sends the first `data request` with byte offset `0` to request the first part of the piece. After receiving a corresponding `data reply`, the peer writes the received data to storage, increments the byte offset with the size of the response data and sends the next `data request`.

When all data of a piece is received, the peer calculates the SHA-1 hash of the downloaded data and compares it to the expected piece hash from the torrent information. Only if these match, should the peer accept the piece and notify others about the newly available piece. If there is a mismatch, the peer assumes that the data was corrupted and retries the piece exchange from scratch. By doing this for every piece, a peer can verify the integrity of the *entire file* and be confident that it has indeed received a bit-for-bit exact copy of the file.

4.5 Multicast piece delivery

With local peer discovery (explained in 3.2), NanoTorrent can already find direct neighbours which are also downloading the same torrent. To fully take advantage of this locality however, peers should also try to efficiently exchange pieces with their immediate neighbours.

This is done by sending `data reply` messages directed to a local peer (identified by a link-local IPv6 address) to *all* local peers as a link-local multicast, rather than a regular link-local unicast. This allows a single peer to distribute a piece to all of its neighbours at once, rather than sending many individual `data reply` messages.

Of course, this means that a peer must be able to handle incoming `data reply` messages for which it is *not* the original requester. If a peer receives data from the start of a piece (byte offset `0`) and it does not yet have a pending request for the sending peer, it can create a new request on the fly and process the incoming data as part of this request. Subsequently received data with other byte offsets can then be handled as part of this newly created request. If the peer already has a different pending request, it cannot create a second request and will have to request it later

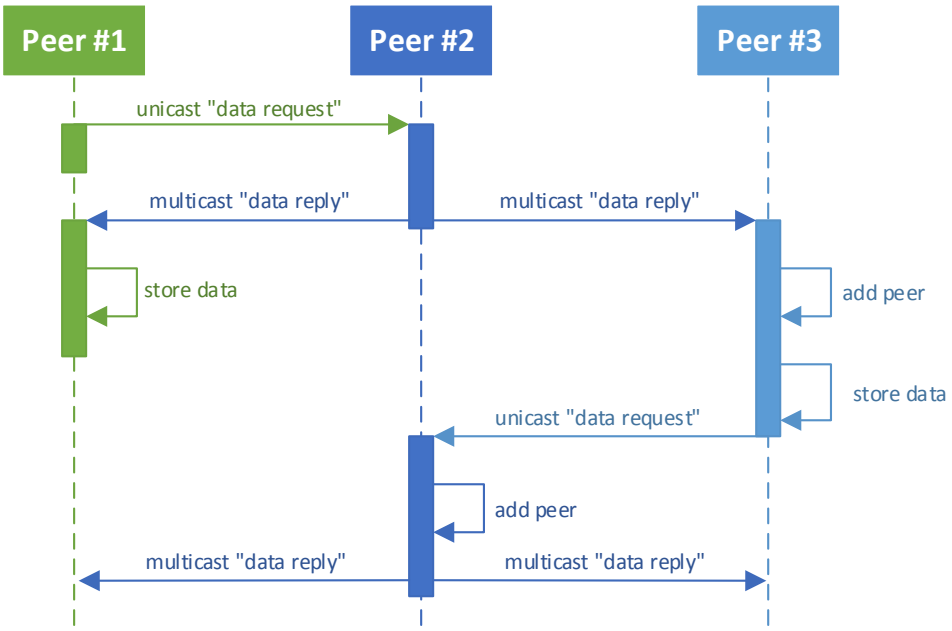


Figure 4.1: Sequence diagram of multicast piece delivery. The multicast `data reply` allows multiple peers to receive piece data, and also act as implicit local peer discovery announcements.

on. This could be improved to allow multiple pending requests, at the cost of more complex request tracking logic.

Figure 4.1 shows an example interaction between three local peers. The first peer has a connection with the second peer, and sends a `data request` for one of the available pieces. The second peer sends the `data reply` as a multicast message to all direct neighbours. The third peer manages to pick up this message from a previously unknown local peer, adds it to its peer connections, creates a pending piece request on-the-fly and stores the received data as part of that request. It then sends a unicast `data request` for the next part of the piece from the second peer. The second peer now discovers this new third peer and again sends a `data reply` to all neighbours. This shows how multiple neighbours can receive data from a single multicast piece exchange, and how such an exchange can also allow two peers to discover each other without a single `have` announcement.

4.6 Conclusion

The file is subdivided into fixed-size pieces, which can be distributed over the P2P network independently. All information about these pieces is contained in the torrent descriptor, which must be delivered to every node in some way before starting the torrent download. The SHA-1 hashes of every piece allow a peer to verify the integrity of every piece and, by extension, the entire file.

Peers try to connect with their discovered peer in order to exchange pieces. They maintain some state about what pieces are available at the other end, and which pieces are currently being requested by connected peers. Peers periodically send `have` messages to keep other peers informed about their download progress, and to indicate that they are still alive.

Pieces are exchanged using a simply request-reply protocol, where one peer sends requests for a part of a piece and the other peer replies with the corresponding data. After all the data of a piece is received, the peer compares the calculated SHA-1 hash with the expected hash from the torrent descriptor. When the verification succeeds, the peer can request its next missing piece and start serving requests for the received piece itself.

To exploit the locality of peers discovered through local peer discovery, peers send their data replies destined to a local peer to *all* local peers using a link-local multicast. Receiving peers attempt to piggy-back onto this piece exchange, allowing a single piece request to serve many local peers. This enables a faster and more efficient distribution inside clusters of nodes.

Chapter 5

Prototype implementation

This chapter discusses the prototype implementations of the tracker and the peer which will be evaluated in chapter 6. These implement the protocol designs from chapters 3 and 4, and provide a working example of the protocol in action.

Section 5.1 introduces the system architecture of the prototype. Section 5.2 goes in-depth about the tracker's implementation, followed by the peer implementation in 5.3. Some limitations of the prototype are outlined in 5.5, ending with a conclusion in section 5.6.

5.1 System architecture

5.1.1 Components

Figure 5.1 shows the types of components making up the NanoTorrent system.

The tracker implementation is discussed in section 5.2. This is responsible for managing the swarms of all torrents, and providing information about these swarms to requesting peers through the NanoTracker protocol. It is assumed to run on a network outside of to the WSN.

The peer implementation is detailed in section 5.3. Peers must join the P2P network, discover other peers, exchange information about available pieces and exchange missing pieces among each other. They communicate with each other through the NanoTorrent protocol.

A border router is needed to allow communications with the tracker since it is located on an external network. This border router provides the bridge between the 6LoWPAN communications on the WSN and the pure IPv6 transmissions on the outside network. It also acts as the 'root' node for the RPL routing protocol used inside the WSN. Since it is a common component in many WSNs platform, it has a standard implementation in Contiki and is used as-is in the prototype architecture.

5.1.2 Communications

Figure 5.2 gives an overview of all protocol communications between these components in a WSN infrastructure.

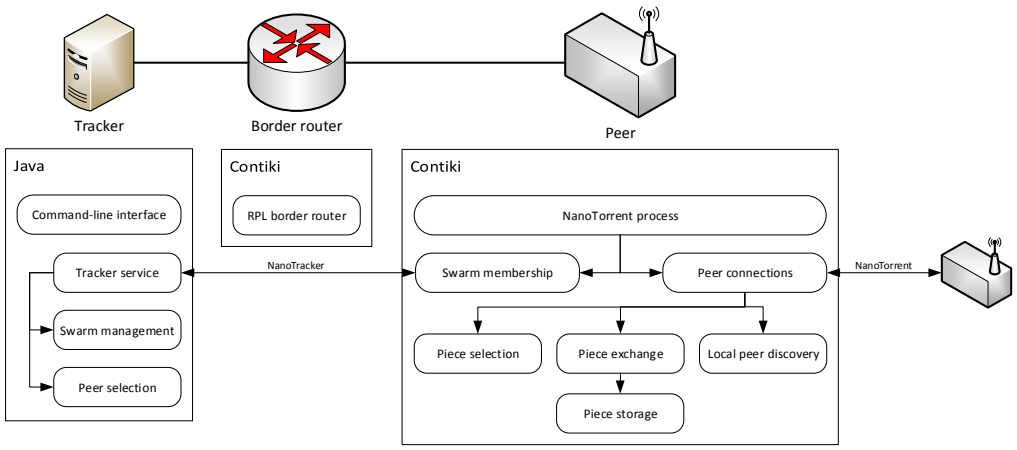


Figure 5.1: NanoTorrent system architecture

Peers start off by sending an announce request (1) to the tracker, which must be routed through the border router. The tracker replies with an announce reply (2) containing a list of peers to connect with.

From there, they can initiate peer connections with these discovered peers by sending an initial `have` message (3). They exchange information about available pieces, and send piece requests for missing pieces (4).

Peers can also discover neighbouring peers through local peer discovery (5), after which they can go through the same process of setting up a peer connection (6) and exchanging pieces (7).

5.2 Tracker

The tracker described in section 3.1 is implemented as a standalone Java application. It starts a service to handle incoming NanoTracker announce requests, and provides a basic command-line interface to inspect its swarms.

Its implementation consists of a tracker service component for communications, a swarm manager to retain stateful information, and a peer selection algorithm to generate the peers lists for announce replies.

5.2.1 Tracker service

The tracker service provides the communication endpoint for peers wanting to join or leave a torrent's swarm. It is responsible for parsing incoming announce requests, triggering the needed operations in the managed swarms, collecting the information needed for the announce reply and then sending the reply.

The service listens on a user-specified UDP port for incoming announce requests, which must be known by all peers wishing to communicate with this tracker. UDP

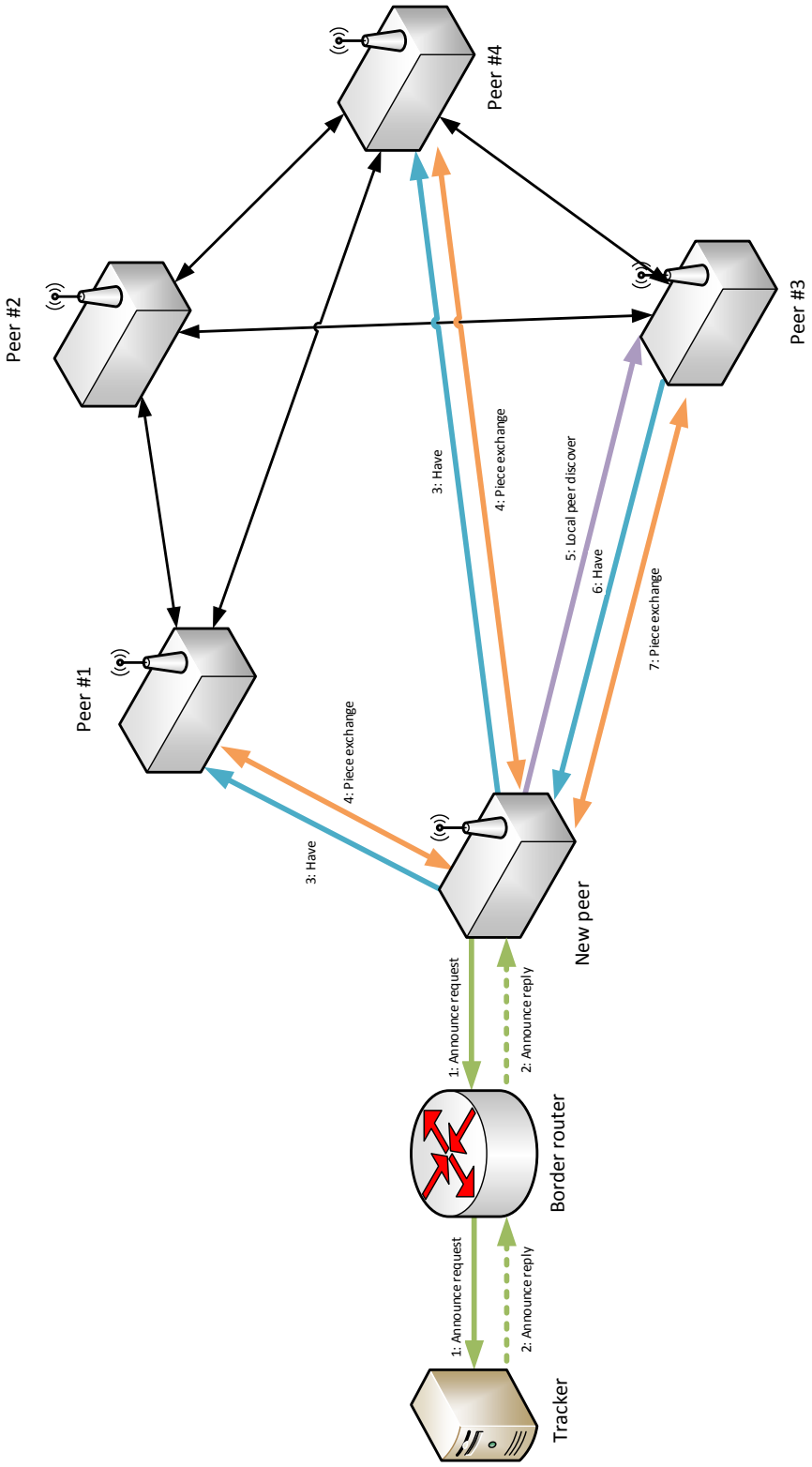


Figure 5.2: Overview of the NanoTorrent protocol communications in a WSN

was chosen as underlying transport protocol rather than TCP, as NanoTracker is a purely request-reply protocol with no need for long-term conversational state.

5.2.2 Swarm management

The tracker must manage stateful information about the torrent swarms that it is tracking. For every swarm, it retains information about the owning torrent (identified by its torrent info hash) and the list of peers currently participating in the swarm. For every peer, the tracker records its IPv6 address, its current membership state and the time of the last received announce request.

When the tracker service receives a request for a torrent with an unknown torrent info hash, it automatically creates a new swarm for that torrent to start tracking it. This is useful for quickly testing the prototype implementation: the tracker can just be started without any additional set up.

After looking up the swarm corresponding to the request's torrent, the tracker looks up the peer in the swarm based on its IPv6 address. If the peer is not yet in the swarm, it is added to it. The tracker updates its view on the peer's membership state using the reported state in the announce request, and refreshes the timestamp of its last announce request.

The tracker uses the timestamp of the last announce request sent by each peer to periodically purge non-responsive peers. When a peer fails to send its periodic announce requests to the tracker, its last announce timestamp does not get refreshed. The tracker can detect this and remove these peers. This ensures that peers will only discover peers that have recently refreshed their swarm state, and are likely to still be online when the peer tries to connect with them.

5.2.3 Peer selection

When the announce request is fully processed and the swarm's state is fully updated, the tracker service must reply to the request with a list of peers from the same swarm. This peers list is then used by the receiving peer to connect with other peers in the swarm.

In the current prototype, the tracker doesn't perform any special selection when generating the list of candidate peers. The tracker simply takes a random subset from the swarm, ensuring that this subset is no larger than the requested number of peers and does not include the requesting peer itself. A more intelligent selection algorithm could take advantage of additional information, for example it could prevent a seed from being connected to another seed (as these would have nothing to exchange with each other) or it could maintain a graph representation of the swarm to ensure that every peer is always directly or indirectly connected to a seed.

The generated peers list is used to construct an announce reply, which is then sent back to the peer by the tracker service.

5.3 Peer

The peer implements the file distribution mechanisms described in chapter 4, the requester's side of the centralized peer discovery from section 3.1 and the local peer discovery of section 3.2. It is implemented in C on the Contiki platform [6], allowing for cross-compilation to many types of WSN platforms.

5.3.1 NanoTorrent process

The main entry point for the peer's implementation is the NanoTorrent main process. This lightweight process initializes all other processes, such as the swarm process maintaining the connection with the tracker and the peer process maintaining connections with other peers. It also reads the list of peers retrieved from the swarm and uses them to initiate new peer connections.

5.3.2 Swarm membership

The swarm membership component is responsible for tracking the peer's membership state, maintaining the connection with the tracker (by periodically sending refreshes) and exposing the retrieved list of discovered peers to the main process.

The client starts a lightweight process to first join the swarm and then send periodic refreshes to the tracker. If it doesn't receive an announce reply after a number of retries, it assumes that the tracker has become unavailable or unreachable and marks itself as having left the swarm.

When the client discovers new peers through the tracker's announce replies, it temporarily stores them in a list while waiting for the main process to try and connect with them. The main process doesn't have to connect with all of the discovered peers, but can let the swarm process hold on to some of them until one of the current peer connections is lost and needs to be quickly replaced.

The main process is notified whenever the membership state changes, e.g. the client has successfully joined the swarm or when the client unexpectedly drops out of the swarm due to connection issues. These inter-process notifications ensure that the main process is aware of the state of all components, and can react when something goes wrong (e.g. losing the connection wit the tracker).

5.3.3 Peer connections

All peer connections and their messages are managed by the peer process. This process maintains a single UDP socket listening on a fixed port (same for all NanoTorrent peers), parses the incoming peer messages and handles them for their various purposes: accepting new connections, responding to piece data request and discovering local peers. It is also responsible for the periodic `have` announcements to all connected peers.

The state that needs to be maintained for every peer connection is described in 4.3, and the used memory structures maintained by the process directly map onto this description. Peers keep track of the 'heartbeat' of every connected peer using

individual timers, which are refreshed on every received message. When a heartbeat timer expires, the peer drops the connection.

In order to keep the protocol scalable as the network size increases, peers limit the number of simultaneously opened peer connections. This prevents peers from opening or accepting too many connections, which all need network traffic and thus battery power to be maintained. The client uses two fixed-size 'pools' for allocating peer connection states: one for incoming connections and one for outgoing connections. Rather than allocating memory for a new connection dynamically on the heap (e.g. using C's `malloc`), these pools are allocated in static memory and have a more predictable behaviour.

When a pool is exhausted, newly requested connections cannot be instantiated and these connections must be closed. However, exhaustion of one pool does not affect the other pool. For example, when a client initiates many outgoing connections with peers discovered locally, it can still accept incoming connections from other peers who discovered the client through the tracker.

5.3.4 Piece selection

Whenever a new peer connection is opened or an existing connection is updated with new information about available pieces, the peer process checks if it can request a missing piece on this connection.

Peers use a rarest-first policy for deciding which piece to select next, as described in 4.4. To implement this, a client tracks the amount of occurrences of every piece at each of its peers, and finds the piece with the least amount of occurrences (excluding zero occurrences). Whenever a peer connects or disconnects, their available pieces are added or subtracted from the list of piece occurrences.

Peers make sure to not request the same piece from two connections at a time. This is managed in a shared variable, which is modified when new requests are made or old requests are finished. However, when a peer has received almost all of the pieces in a torrent, it is not desirable having to wait on just one peer to deliver the last piece. In this case, the peer goes into what BitTorrent calls *end-game mode*: when nearing completion, a single piece may be requested from more than one connection. The impact of this is minimal as it only occurs at the very end of the download, but makes sure that peers can quickly transition from almost done to fully seeding.

5.3.5 Piece exchange

When a peer discovers an interesting piece at another peer, it can request this piece and start downloading it as detailed in section4.4.

Piece requests are managed individually per connection, with different retry counters and timers for the multiple outstanding requests. This allows parallel requests to operate independently from each other, preventing a single failing connection to take down all connections.

Piece replies destined to local neighbours are sent as multicast packets rather than unicast packets, as specified in 4.5. Since the only change is in the destination IPv6 address, this is readily implemented as a small special case.

5.3.6 Piece storage

The piece data received in piece data replies is written to a file in the Contiki File System (CFS). The client uses the piece index and the piece data offset to calculate the byte offset in the file, and writes the bytes from the reply into the file. Similarly, data is read from the file when sending a piece data reply.

On the AVR Zigduino platform, Contiki File System (CFS) is configured to use the EEPROM for file system storage. This allows for 4 KB of read/write storage [7], which is sufficient for most network reprogramming or reconfiguration use cases. For experiments in the COOJA simulator [21], CFS uses an in-memory file system.

5.3.7 Local peer discovery

Peer discovery through local multicast as discussed in section 3.2 is implemented as a modification to the periodic `have` announcements. Rather than sending the `have` announcements to individual local peers, it is multicasted to all local neighbours.

Incoming `have` multicast announcements are handled as regular incoming connection requests. The peer will try to accept the connection (if it hasn't already), and will process the message if it has a connection. As noted in 4.3, peers must not reply with a `close` message if it fails to accept a connection in response to a multicast `have`. Other than that, it can treat local peers and remote peers uniformly.

5.4 Usage

5.4.1 Deployment

In order to deploy NanoTorrent onto a WSN, the network operator must deploy the tracker on a computer on an external network, deploy a Contiki application using NanoTorrent onto the WSN nodes and connect the external network with the WSN with a border router.

Tracker

The tracker is a simple Java command-line application which can be started to operate on any UDP port. The computer on which it is deployed must have a Java runtime installed and must have a connection with the border router. Other than that, the tracker is very much 'fire and forget': it can run in the background and doesn't need user interaction.

Peer

The prototype is implemented in C for the Contiki platform, so it must be compiled for the particular platform used by the WSN nodes. The implementation does not rely on hardware-specific features, so it can readily be cross-compiled to any of Contiki's supported platforms – although it is recommended to fine-tune the protocol's configuration to the specifics of the used platform.

The prototype comes with a demo application, which has been verified to run on both the AVR Zigduino platform and in the COOJA simulator environment. The demo is implemented in `peer/demo.c` and can be compiled with the Makefile rule `demo`.

Border router

The RPL border router is a standard component in Contiki, and can be found in `contiki/examples/ipv6/rpl-border-router`.

5.4.2 Bootstrapping

Before peers can start downloading a torrent, they must have received the torrent descriptor from the distributor, as explained in 3.1.2. The torrent descriptor is needed to 'bootstrap' the NanoTorrent client, as it contains vital information needed for the file distribution. In the case of BitTorrent, the distributor creates and publishes a `.torrent` file containing the torrent's metainfo, as mentioned in section 2.1.1. Users download this `.torrent` file and open it with a BitTorrent client to start the download.

NanoGen toolchain

The distributor can use the included NanoGen tool to generate a torrent descriptor file to be distributed to all destined NanoTorrent nodes. NanoGen takes as input the file to be distributed, details about the tracker and the piece size and produces a binary file which can be parsed by the NanoTorrent prototype implementation. The tool takes care of subdividing the input file into pieces of the given size, calculating the SHA-1 hashes of every piece and writing them to the output file.

To inspect the contents of a generated torrent descriptor file, the NanoRead utility was added. This utility takes a torrent descriptor file as input and prints its contents in a human-readable format. This is useful for verifying the contents of a generated or download torrent descriptor, and for debugging purposes.

NanoGen and NanoRead are implemented in C as Contiki applications in `peer/nanogen.c` and `peer/nanoread.c`. They can be compiled through the Makefile rules `nanogen` and `nanoread` with the flag `TARGET=native` to compile for the native platform (e.g. Linux). These rules produce executables for the command-line applications named `nanogen.native` and `nanoread.native`.

Torrent descriptor distribution

The current prototype implementation does not provide a mechanism for distributing a torrent descriptor. Like BitTorrent, the NanoTorrent client requires that the descriptor is available locally on the node, and is not concerned with *how* the descriptor is retrieved.

For evaluation and demonstration purposes, the torrent descriptor is hard-coded into the NanoTorrent image at compilation time. The Makefile rule for the demo application takes a `TORRENT_DESC` argument which specifies the path to the torrent descriptor file. This file is written in the static file system section of the created demo image, which will be read by the demo application at node start up.

Seed configuration

In order to configure a node act as a seed, it must have the whole file available at start up. NanoTorrent clients use the Contiki File System (CFS) for storing the distributed file, so the seed must have the file prepared in its local file system.

On the AVR Zigduino platform, Contiki File System (CFS) is configured to use the EEPROM for file system storage. The Makefile provides the `demo.seed` rule to generate an EEPROM image for the file system with the given `TORRENT_FILE` at the start of the CFS region. A seed can then be flashed with the `demo.seed.eu` and `demo.seed.u` rules.

For evaluating the NanoTorrent implementation in the COOJA simulator [21], the distributed file must be uploaded to the appropriate NanoTorrent nodes through COOJA's mote configuration interfaces.

5.5 Limitations

The current implementation is very much just a prototype. Many features common in mature file distributions are missing or are not fully fleshed out.

5.5.1 Bootstrapping

Currently, the torrent descriptor is hard-coded into the peer's program during compilation, locking the peer into downloading just one file. A more complete network reprogramming solution based on NanoTorrent should include a more versatile distribution mechanism for this descriptor.

The particularities of such mechanism depend heavily on the used WSN middleware platform. For example, a WSN running on the LooCI [15] platform would communicate mostly through LooCI's event-based communication network. A reconfiguration engine based on NanoTorrent for LooCI could send the torrent descriptor as payload data of a LooCI event, and have them launch the NanoTorrent process in response.

5.5.2 Security

The prototype assumes that nodes in the network can be trusted. Although this assumption is defendable in the context of experimental WSNs, it does not hold for large-scale deployments where malicious peers could attack the network from the outside or the inside.

For example, a malicious node could intentionally send corrupted piece data to a peer, causing it to write the wrong data to its file. The peer will detect the corruption only after it has fully received the piece (possibly through multiple data requests), and will need to request the same piece again.

The tracker is also susceptible to uncontrolled announce requests. Every time a new request arrives for an unknown torrent, the tracker creates a new swarm and starts tracking it. This is sufficient for small-scale experimental uses, but could be abused by sending many announce requests with bogus torrent info hashes. The tracker would allocate a lot of swarms which are never used and never cleaned up, potentially crashing it due to it running out of memory. To prevent such abuse, trackers deployed in the wild should only accept announce requests for known torrents registered in a backing database.

5.6 Conclusion

The prototype for the tracker and the peer provides the basic implementation of the protocol designs from chapters 3 and 4, taking into account the constraints and limitations imposed by WSN nodes. It comes with tools for generating and inspecting torrent descriptors, monitoring the swarms on the tracker and compiling images for NanoTorrent leechers and seeds.

The prototype lacks additional features commonly found in more mature file distribution mechanisms, such as an easy-to-use bootstrapping process or security checks. However, it provides enough to do useful evaluations of the proposed protocol in small experiments.

Chapter 6

Evaluation

This chapter evaluates the prototype implementation of the NanoTorrent protocol.

Section 6.1 lists the formed hypotheses that need to be evaluated. Section 6.2 describes the evaluation methodology. Section 6.3 evaluates the scalability of the protocol, and section 6.4 tests the protocol in a heterogeneous deployment. Section 6.5 lists possible improvements that could be made to the protocol, and 6.6 concludes with a discussion.

6.1 Hypotheses

The goal of NanoTorrent is to deliver files to many WSN nodes in a reasonable amount of time, while remaining efficient on battery usage. It should be able to achieve this for many network topologies, which may possibly consist of different types of nodes – not all of them running a NanoTorrent implementation.

To allow for both fast distribution and heterogeneity in the network, NanoTorrent approaches this problem with a P2P file distribution protocol supported by a hybrid peer discovery mechanism. The use of both local peers and remote peers should allow for fast distribution among neighbouring nodes, while still enabling piece exchange with faraway nodes which can help spreading the file in other parts of the network.

6.2 Methodology

To test these hypotheses, the NanoTorrent implementation is run in various experiment configurations inside the COOJA simulator. In each experiment, nodes are tasked to distribute a given torrent and must fully retrieve and verify the contents of the distributed file.

6.2.1 AVR Zigduino

The prototype has been tested and verified in a small network with AVR Zigduino nodes consisting of one border router, one initial seed and three regular peers. The border router is connected to a laptop running the tracker implementation.

Although this set up works, it does not allow a very thorough evaluation of the protocol's capabilities to scale up to larger networks.

6.2.2 COOJA simulator

The evaluations were performed in the COOJA simulator [21]. This is a simulation environment for Contiki nodes that emulates the hardware layer of a Contiki platform. It allows Contiki applications to be cross-compiled with the COOJA-specific hardware implementation, and can run these specially compiled versions in various simulated conditions.

COOJA comes with the Contiki distribution and is implemented as a Java application. The simulator interface allows the user to add new motes, place them in a simulated wireless network and configure them. While running a simulation, COOJA shows and records the message logs of all nodes, the timeline of occurred events and the packets transmitted across the network. The recorded information can also be exported for in-depth analysis by other tools.

COOJA simulates the platform and the network at the hardware layer, and interfaces with the compiled C code through the Java Native Interface (JNI). This allows COOJA to run almost any Contiki application, as it behaves just like any other target Contiki platform for cross-compilation. This method of full system simulation is more faithful to real platforms than other simulators such as TOSSIM [19] which can only simulate one layer of the platform.

6.3 Scalability

To be usable in both small-scale experimental setups and large-scale WSN deployments, NanoTorrent must be able to scale as more nodes are added to the network.

6.3.1 Set up

Figure 6.1 shows the simulation set up. In each experiment, N^2 nodes are deployed in an $N \times N$ grid layout such that each node has at most 4 neighbours. Every node has a torrent descriptor for a 2 KB file consisting of 8 pieces of 256 KB. The top left node acts as the initial seed from which this file will start propagating over the network. This node is also connected with a border router, which connects the WSN with the tracker. The experiment was carried out for $N = 3$ to 7, resulting in networks consisting of 9 to 49 NanoTorrent peers.

Since all nodes in this set up download the same torrent, file distribution could also be done *without* a centralized tracker. To analyse the benefits or downsides to this approach, every set up is tested once without the tracker and once with both peer discovery mechanisms enabled.

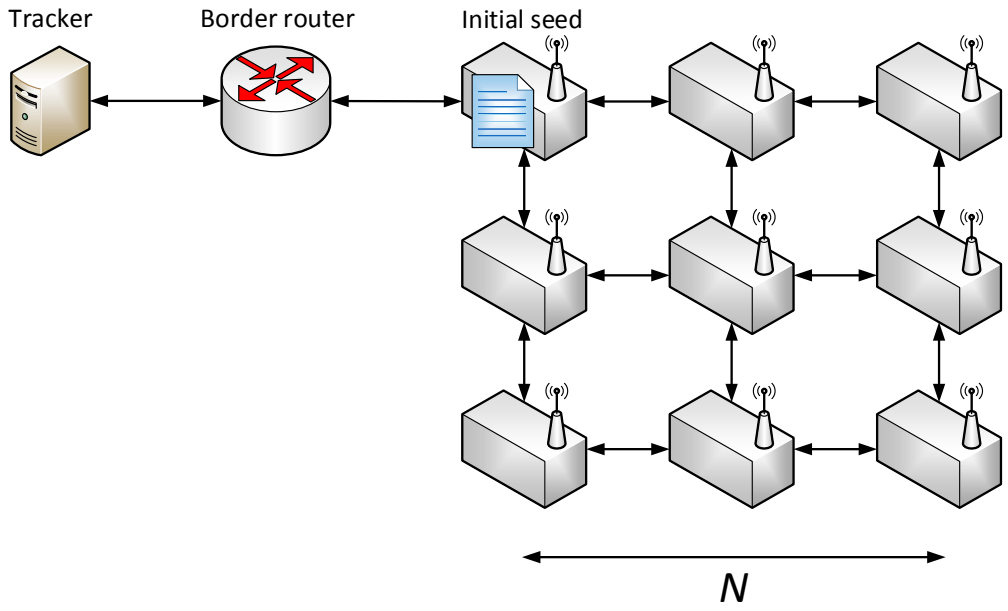


Figure 6.1: Network set up for scalability experiment. The WSN consists of $N \times N$ nodes in a grid layout, with one initial seed.

6.3.2 Results

Figure 6.2 shows the deployment times (i.e. the time to deploy the file to all nodes) for the different network diameters, both with and without the tracker.

Figure 6.3 shows the total number of transmissions sent over the network. This total is broken down into the different upper-layer network protocols in figure 6.4. NanoTorrent and NanoTracker indicate messages between peers and with the tracker. IEEE 802.15.4 messages consist of link-layer acknowledgements for transmitted frames. ICMPv6 messages are used to maintain the IPv6 and RPL routing mechanisms. 6LoWPAN messages consist of fragmented packets which are later re-assembled into larger IPv6 packets.

Figure 6.5 gives a more in-depth analysis of the download progression in a 3×3 network. 8 nodes (numbered 3 to 10) receive the file from a single initial seed.

Figure 6.6 displays the completion times of peers in a 7×7 network. Rather than showing the individual progress of each of the 49 nodes similar to figure 6.5, this figure only shows the time at which they receive their last missing piece, grouping the results in 5 second intervals. The initial seed is also shown as the only peer completing the torrent after 0 seconds.

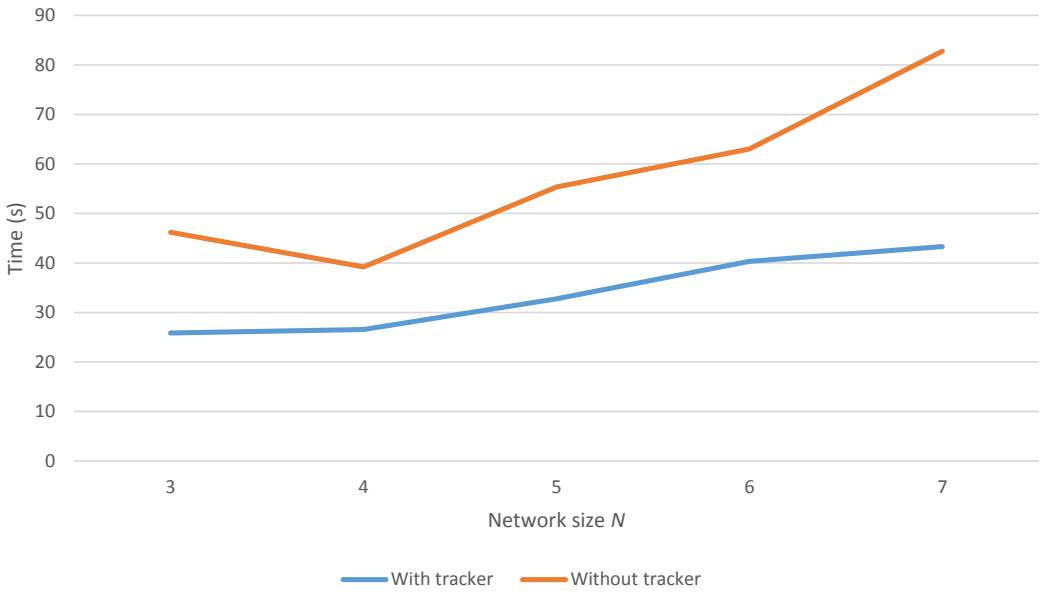


Figure 6.2: Time to deploy the full file to all nodes for network diameter N.

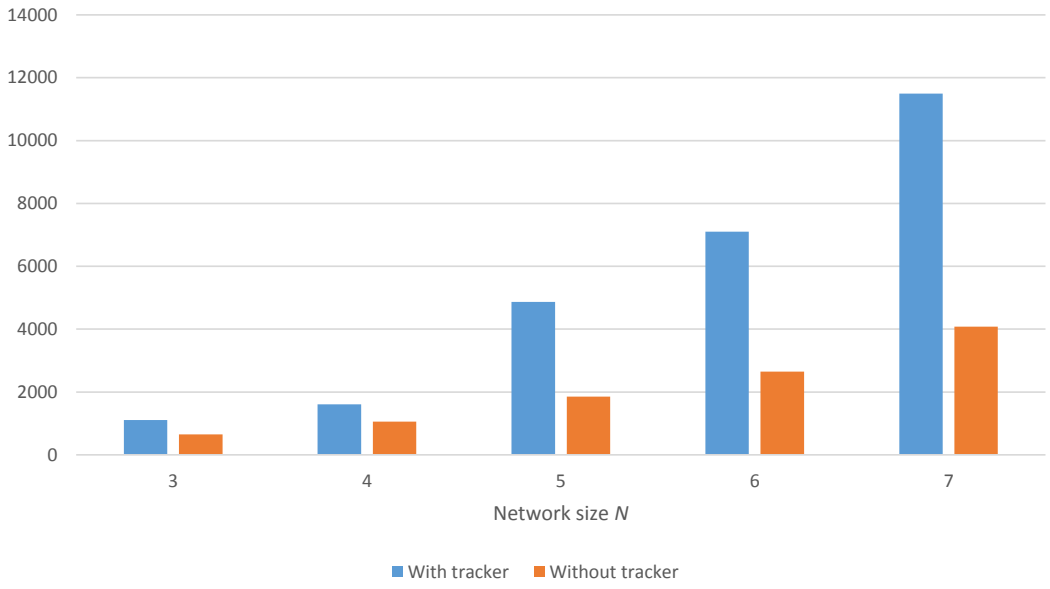
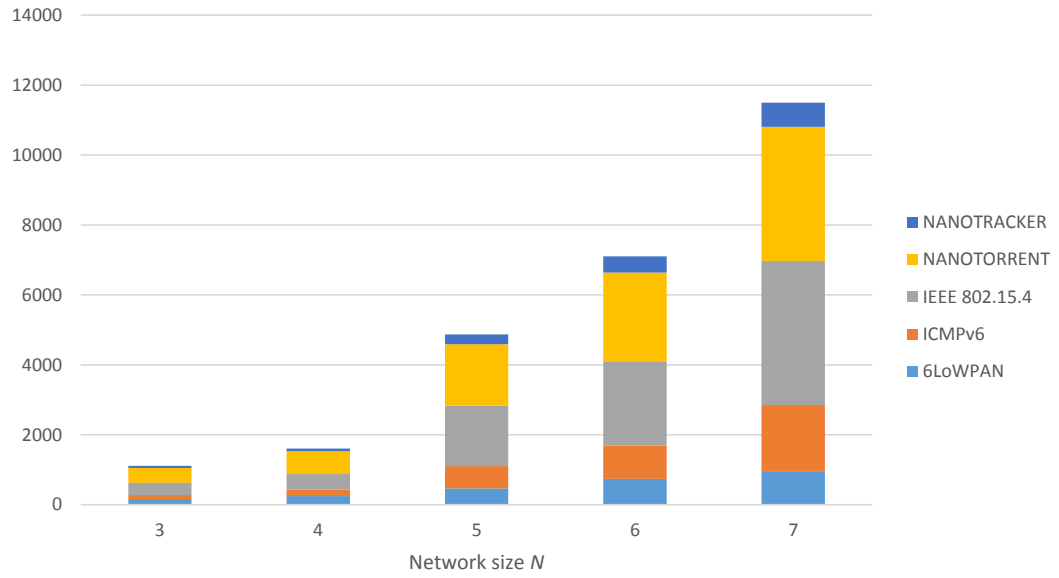
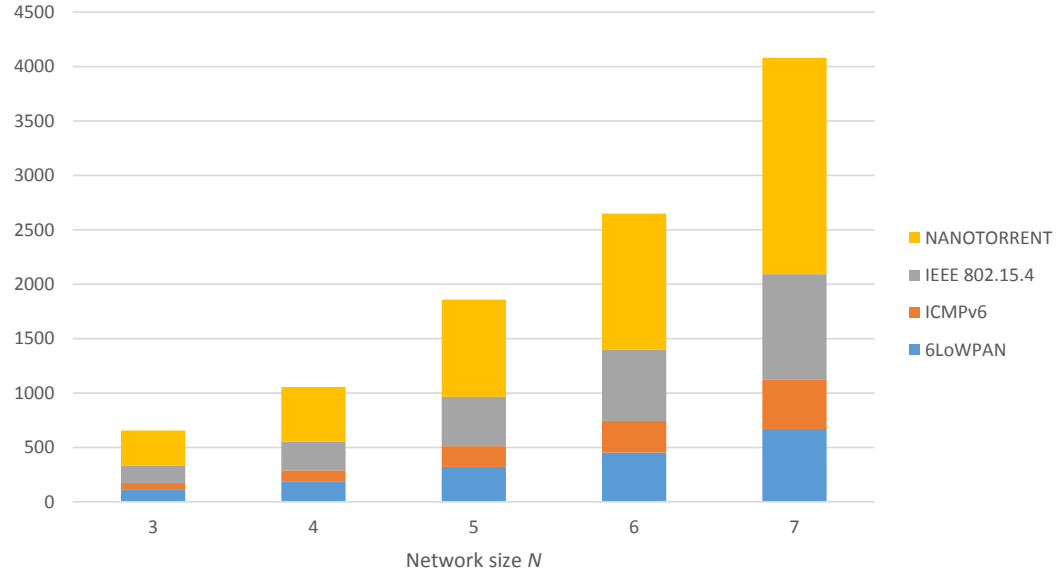


Figure 6.3: Total transmissions sent during deployment for increasing network diameter N.

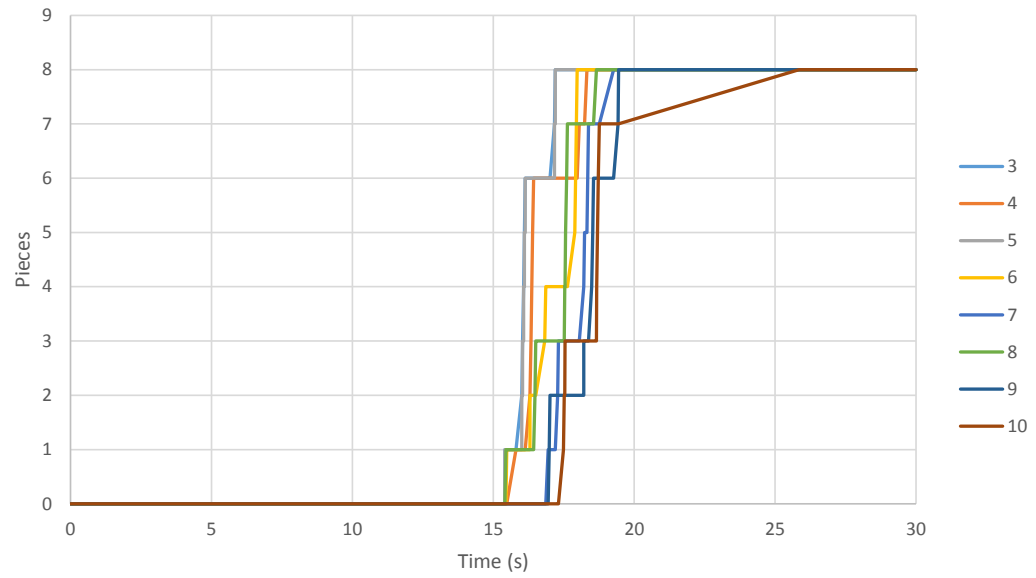


(a) With tracker.

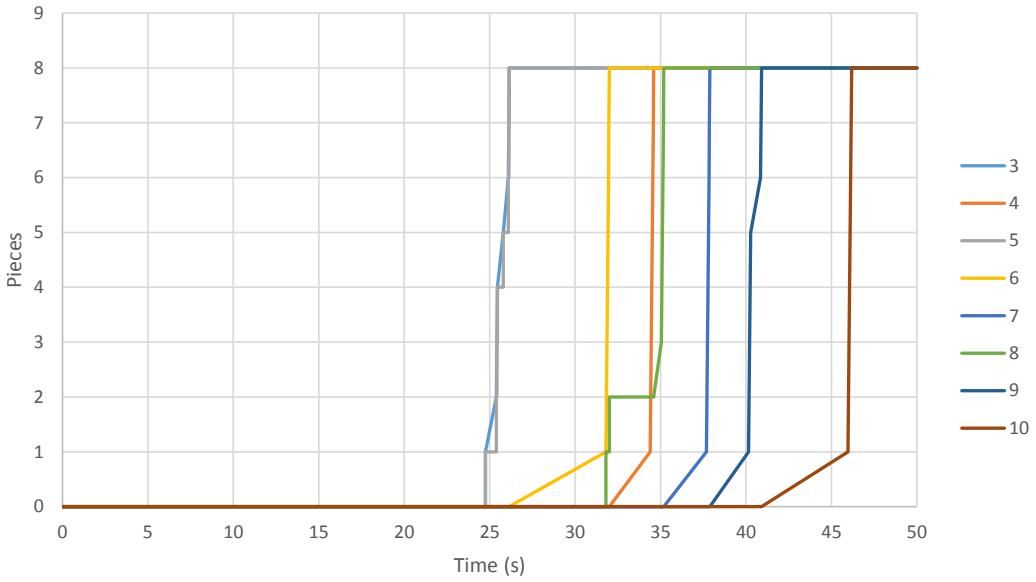


(b) Without tracker.

Figure 6.4: Protocol breakdown of transmissions in $N \times N$ network

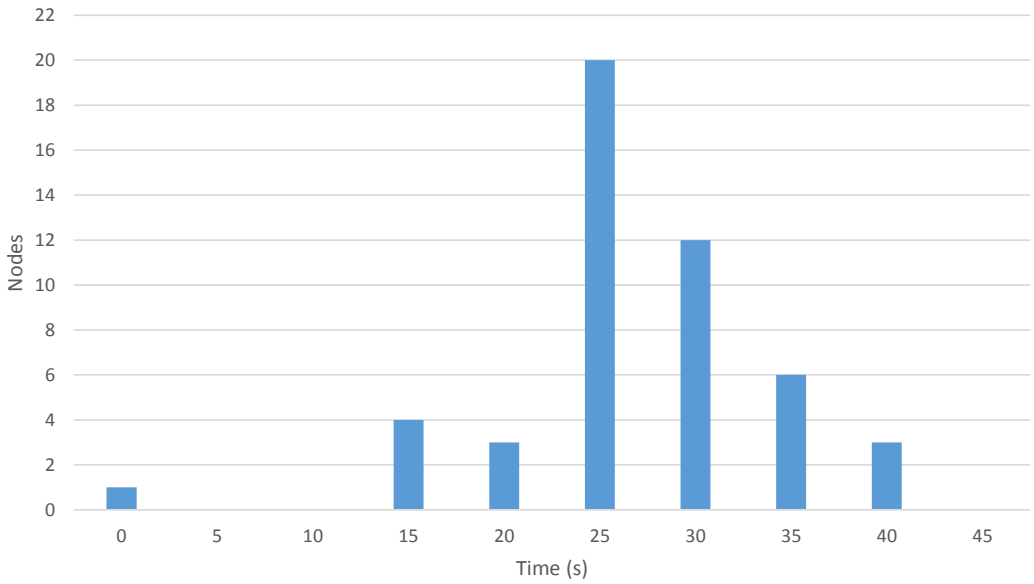


(a) With tracker.

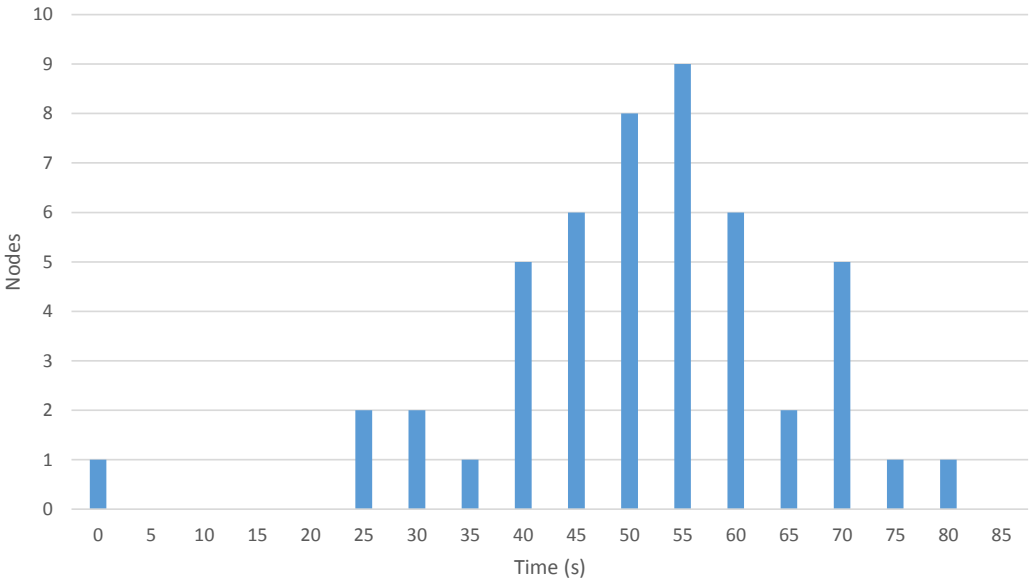


(b) Without tracker.

Figure 6.5: Download progression for 8-piece file in 3×3 network. The different coloured lines indicate the progress of different peers.



(a) With tracker.



(b) Without tracker.

Figure 6.6: Completion times in 7×7 network. The labels on the X-axis denote the start of an interval, e.g. $0s$ to $5s$.

6.3.3 Interpretation

Deployment time

Figure 6.2 shows that the use of a tracker clearly decreases the total time needed to deploy the file to the whole network. Every peer has additional connections with remote peers from which they can request pieces. Peers on the edge of the network can connect with peers closer to the initial seed, acquire pieces faster and help distribute them to their neighbours.

Without a tracker, every peer must wait until one of its neighbours starts receiving the file. Pieces propagate from the initial seed to the rest of the network as a 'wave'. This means that peers on the far edge of the network barely participate in the file distribution, only receiving the file when all other peers have already received it.

Both approaches appear to scale linearly with network size in terms of deployment time. As the network increases in size, the path length between the initial seed and the furthest peer increases linearly as well which is the primary contributor to the deployment time.

Transmissions

However, the increased deployment speed from using a tracker comes at a cost in terms of transmitted messages as shown by figure 6.3. As peers connect with more peers, they need to send more periodic announcements and reply to requests from more peers. While peers on the edge of the network can help balance the load from distributing pieces, communicating with faraway peers is more costly to the network as a whole. The messages between a node and the tracker or its remote peers must traverse the network, taking multiple hops along intermediate nodes which need to use their transmission power to route the IPv6 packets. These long distance messages account for a lot of overhead transmissions when compared to the tracker-less approach, and this overhead increases with network diameter.

When breaking down these transmissions by protocol, most messages show up as either NanoTorrent or link-layer acknowledgements. The link-layer acknowledgements are an integral part of the network stack, and their usage increases as messages take more hops to get to their destination. This increase is more prevalent in the experiments with a tracker, as NanoTorrent messages need to routed by more nodes to reach a remote peer.

While the number of transmissions without a tracker scales fairly linearly with increasing network size, the number of transmissions when using a tracker seems to increase more than linearly. The additional connections opened by peers cause extra messages on the network which need to be routed across multiple hops.

Progression

In the 3×3 network of figure 6.5a, all nodes quickly start receiving the file as soon as they receive the first periodic `have` announcement from the initial seed. Thanks to the tracker, multiple nodes are connected to this initial seed, which means more peers

benefit from this initial announcement. In figure 6.5b, the absence of a tracker results in only the two direct neighbours of the seed (nodes 3 and 5) receiving this first announcement. The initial seed communicates only with these two neighbours, which in quick succession request all pieces of the file. However, they take a couple of seconds before notifying their own neighbours, and have already completely downloaded the file before their next `have` announcement is due. This leaves some room for improvement for fine-tuning the timings of these announcements, to let neighbours more quickly know when new pieces are available.

Figure 6.6 shows the distribution of the completion times of all nodes in a 7×7 network. For this scenario, the use of a tracker results in a nearly two-fold speed up, distributing the last piece after just 43 seconds as opposed to 82 seconds in the tracker-less experiment. Most peers complete the download around the halfway point, when the 'wave front' of the file distribution is the widest at the main diagonal of the grid. In theory, this is when the $N - 1$ nodes on the sub diagonal of the square grid are sending pieces to N nodes on the main diagonal, reaching the maximum number of concurrent transfers. When a tracker is employed, the distribution is more skewed as peers use remote connections to exchange pieces.

6.4 Heterogeneity

Whereas other distribution protocols for WSNs such as Deluge (see 2.2.1) require all nodes to run the same distribution protocol, NanoTorrent can operate in a *heterogeneous* network consisting of nodes with different tasks and different programs. As long as all nodes properly route IPv6 traffic across the WSN, NanoTorrent peers can communicate with each other even if they are separated by non-NanoTorrent nodes.

6.4.1 Set up

To validate this mode of operation, an experiment was carried out with two clusters each consisting of 5×5 NanoTorrent peers, which are only connected by a single border router node. This border router routes IPv6 traffic between the two clusters and to the tracker on the external network. Peers in different clusters can only reach each other through this border router. The set up is shown in figure 6.7.

6.4.2 Results

Figure 6.8a show the download progress of 5 nodes from each cluster of NanoTorrent nodes. Nodes numbered 2 to 26 are in the cluster with the initial seed, while nodes 27 to 51 are in the other cluster.

Figure 6.8b displays the completion times of all the peers in the network, i.e. the time at which a peer receives their last missing piece, grouping the results in 5 second intervals. The initial seed is shown as the only peer completing the torrent after 0 seconds.

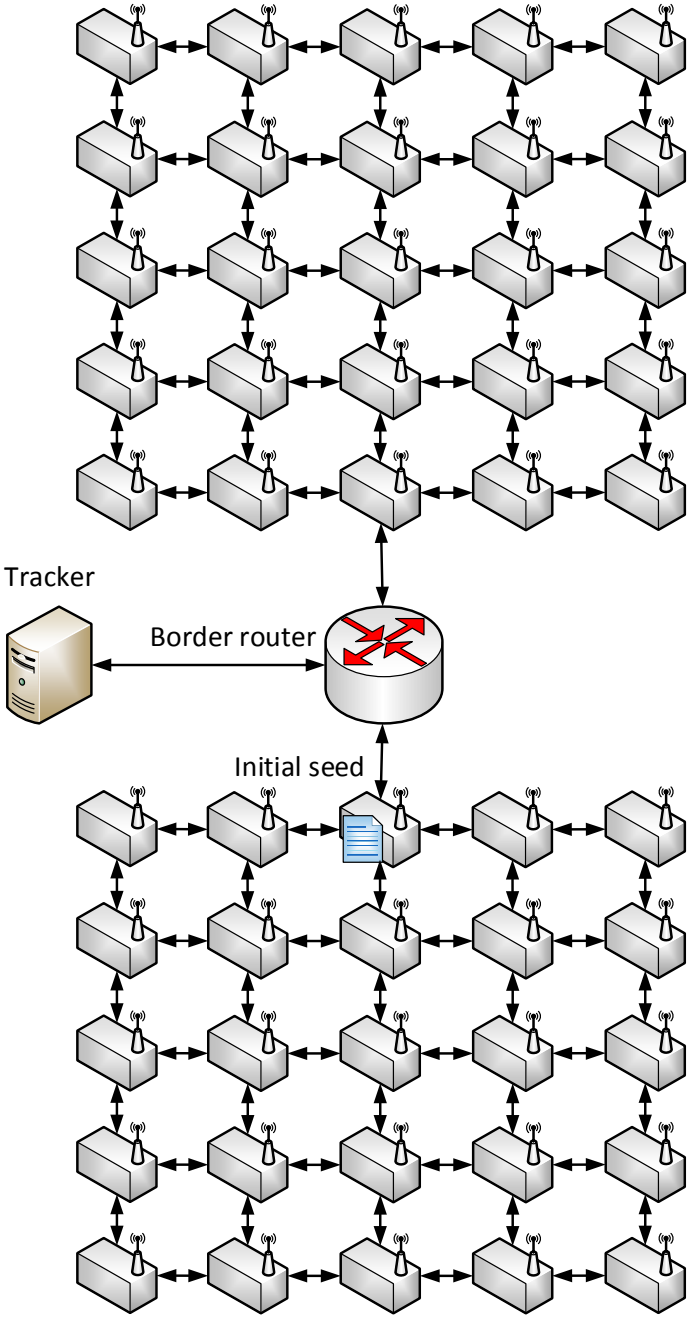
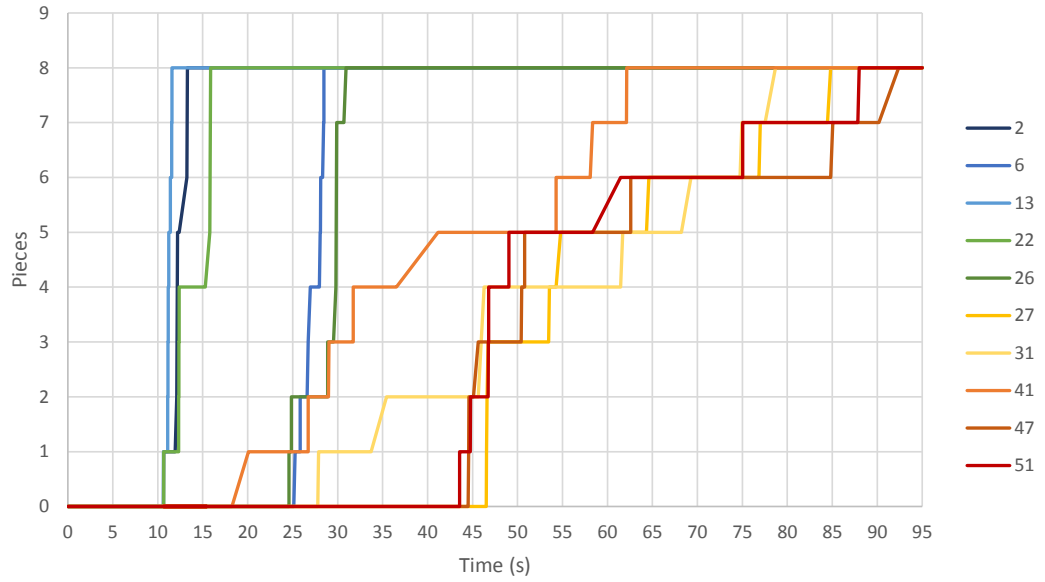
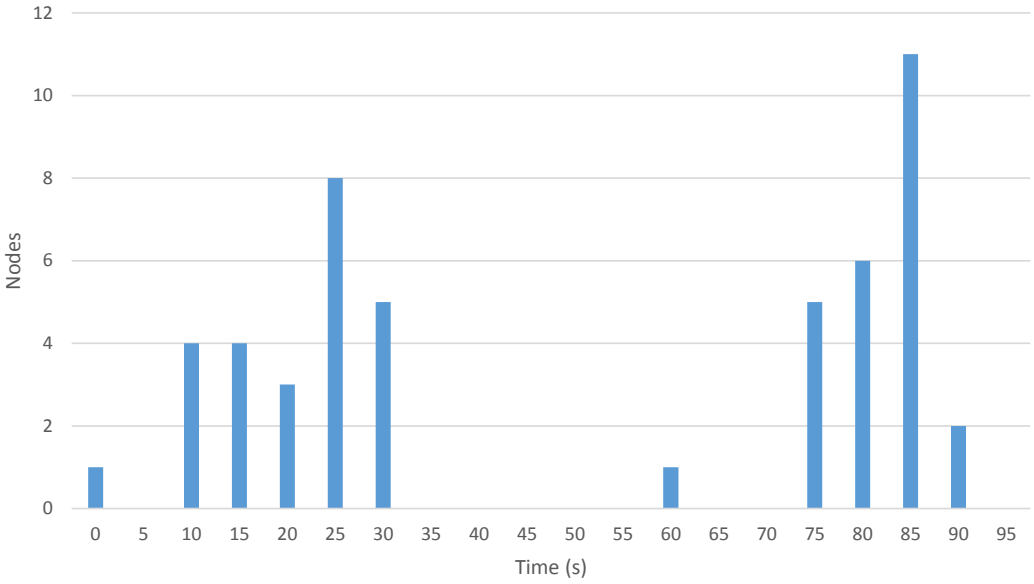


Figure 6.7: Network set up for Heterogeneity experiment. The WSN consists of 2 separate clusters of 5×5 nodes in a grid layout, with one initial seed in one of the clusters.



(a) Download progression of a selected number of nodes.



(b) Completion times.

Figure 6.8: Results for a network consisting of two 5×5 clusters.

6.4.3 Interpretation

As expected from the results in section 6.3, figure 6.8a shows that peers in the cluster with the initial seed start exchanging pieces quickly. After 35 seconds, the whole cluster has received all pieces of the file and start seeding the torrent.

In the meantime, some nodes in the other cluster manage to connect a peer in the seeding cluster through the tracker and slowly start receiving pieces. Their initial progress is slowed down by the need to pass through the single border router, but quickly start spreading the received pieces to other peers in their cluster. It takes a minute for the first node in this cluster to complete its download, and 30 seconds later the last node receives its last piece.

This slow progression in the second cluster is evident in the distribution of completion times in figure 6.8b. The last node in the seeding cluster completes its download a full 30 seconds before the first node in the other cluster does. Although these nodes do progress, they are fully dependent on the seeding cluster and they are limited by the network.

6.5 Possible improvements

There is definitely room improvement for the NanoTorrent protocol. Configuration parameters still need to be tuned, multiple peer discovery mechanisms may discover the same peer multiple times, and piece data transmissions can still be made more efficient.

6.5.1 Parameter tuning

The protocol uses many timers to coordinate its behaviour, most of which still need to be fine-tuned. For example, file distribution only takes off after all nodes are properly initialized and the initial seed sends its first periodic `have` announcement – currently after 15 seconds. These 15 seconds at the start of the distribution is time wasted doing nothing useful, delaying the network reprogramming and the return to normal network operation. Increasing the frequency of `have` announcements for example can speed up deployment, but also adds extra load on the network. Another solution could be to send the initial announcement already after e.g. 5 seconds, enough to let other nodes initialize, and then revert to a longer period.

There is currently very little coordination between neighbouring nodes when sending new piece requests. In Deluge (2.2.1), nodes listen for messages by others and suppress their own transmission if they hear identical messages being sent by other neighbours. This could be adopted by NanoTorrent for local piece exchange, decreasing the amount of redundantly sent piece requests and replies. Remote peers discovered through the tracker however cannot hear these local neighbours, so they should still be treated separately.

6.5.2 Duplicate peer connections due to dual peer discovery

When a peer uses multiple peer discovery mechanisms, it may be possible that it discovers the same peer through different mechanisms. This can lead to peers having one or more duplicate peer connections, where the same peer is at the other end of multiple connections.

In the current implementation, peers identify other peers by their full IPv6 address. Peers discovered by different mechanisms never have the same address: local peer discovery only discovers peers with a *link-local* address, while the tracker will only find peers with a *global* address. This means that if the same peer is discovered through both mechanisms, it will be seen as two different peers with two different addresses.

In BitTorrent, this issue is resolved by having peers randomly generate a peer ID of 20 bytes [4]. This is longer and likely more random than a 16 byte, rigidly structured IPv6 address. Peers must identify themselves with this ID through all peer discovery mechanisms, allowing others to detect duplicate peers with great confidence.

A possible solution for NanoTorrent would be to only use the *interface identifier* of the IPv6 address as peer identifier. WSN nodes generally use stateless address autoconfiguration [9] to generate this interface identifier, and re-use it for all networks. On the Internet, this is not very desirable as it can function as a 'supercookie' to track a user's activity on any website. However, for the purpose of identifying a single peer in multiple networks or address scopes, this may prove to be a reliable identifier. When a peer discovers a remote peer whose address has the same interface identifier as one of its link-local peers, it can assume that it already has a connection with that peer and instead connect with a different remote peer.

6.5.3 Pipelined piece data replies

NanoTorrent appears to use a lot of transmissions to distribute a file. Some of this can be contributed to the hybrid peer discovery mechanism, but other parts can also still be improved. For example, when a peer receives an piece request, it currently replies with just one piece data reply, waiting for the requester to receive this and send the next request.

Rather than waiting for this round-trip, the peer could assume that the requester will want the following parts of the data as well. It could schedule the next part to be sent some time later, saving the requester from having to send requests.

However, this moves the burden of maintaining state about a piece exchange from the requesting peer to the transmitting peer. Now, the transmitter must track which peers have requested one of their pieces, schedule transmissions and manage retries. Moreover, this may become difficult to combine with multicast piece delivery, where multiple local peers are served by a single piece data reply. This trade-off could be evaluated in a variation on the protocol.

6.5.4 Impact of fragmentation on piece delivery

When peers receive a data request for one of their available pieces, they will construct a data reply and fill it with as much data as they can. In the current prototype implementation, the size of a data reply is limited by the node's IPv6 packet buffer size, which allows for 164 bytes of piece data to be sent at once by a COOJA simulator node.

However, the constructed packet may become fragmented by the 6LoWPAN layer to accommodate the link-layer maximum transmission unit (MTU). 6LoWPAN splits up a single IPv6 packet into multiple fragmented 6LoWPAN packets, and re-assembles them at the other end.

This fragmentation at two different layers may prove to be decremental to the protocol's performance. A possible solution would be to adapt NanoTorrent's configuration to either make sure all data replies fit in a single 6LoWPAN packet, or it could be changed to send the whole piece as a single over-sized IPv6 packet and let 6LoWPAN deal with all fragmentation. The impact of such a change is currently unknown, and needs more investigation.

6.6 Discussion

From the results of 6.3, it turns out that using a centralized tracker to allow faraway peers to connect with each other comes with a trade-off. One the one hand, the file can be distributed much faster, since nodes at the edge of the network also help balance the load of sending pieces. On the other hand, messages between distant nodes need to be routed by many intermediate hops which need to spend additional energy to route these messages. This impacts the scalability of the protocol to larger networks, whereas the tracker-less approach keeps its transmissions low.

Whereas other distribution protocols such as Deluge can only work in homogeneous networks consisting of a single type of node (see 2.2.1), NanoTorrent's use of IPv6 allows for file distribution in heterogeneous networks consisting of separate clusters of peers. In the experiment from 6.4, the initial seed can be placed in one cluster, and peers from this seeding cluster can help distribute the file to remote peers in the other cluster by using the tracker. Progress in this second cluster is slowed down by the need for multi-hop routing, but quickly ramps up once most pieces become locally available in the cluster and can be distributed through local communications.

The protocol can still be improved upon to increase the performance and reduce the amount of needed transmissions. However, the impact of these suggested improvements is still unknown and could come with additional trade-offs of their own. There is still a lot left to be researched in this area.

Chapter 7

Conclusion

This chapter concludes this master's thesis. Section 7.1 summarizes the contents of the text, section 7.2 looks at the lessons learned from NanoTorrent, and 7.3 discusses potential future work on this topic.

7.1 Summary

7.1.1 Introduction

In order to keep wireless sensor networks operating for long periods of time, they must be able to adapt and evolve even after deployment. Nodes must be able to receive 'over-the-air' updates, where new program files or configurations must be distributed over the WSN and received by all destined nodes so they can reconfigure themselves. This must be done fast, efficient and load balanced over the whole network.

NanoTorrent takes a peer-to-peer approach to the problem, where nodes act as peers and download and upload pieces of the file to each other. The main contribution of this protocol is the exploration of a hybrid peer discovery mechanism, combining discovery through a centralized tracker with local neighbour discovery. This allows NanoTorrent to function in more heterogeneous WSN deployments, where not all nodes run the same program or are located on different networks.

7.1.2 Related work

The protocol design is heavily inspired by BitTorrent, a popular P2P file distribution protocol for the Internet. It borrows concepts such as torrents, trackers and pieces, but adapts them to the peculiarities of WSNs. BitTorrent also has an extension to discover local peers on the same LAN through local multicasts, but this is currently underused. NanoTorrent explores this possibility with its own local peer discovery that is similar in design, but is more deeply integrated in the P2P protocol.

Other P2P protocols exist for file distribution in a WSN. Deluge uses local broadcasts to propagate the file over the network, and avoids redundant transmissions

by listening for identical messages from others. TinyTorrents is also influenced by BitTorrent, and supports interoperability with BitTorrent through a gateway.

The researched P2P solutions for WSNs do not take advantage of the IPv6 architecture which is becoming more commonly available in many WSN networks. This opens an opportunity for a file distribution that utilizes the multi-hop connectivity between distant nodes in the network and can even communicate with outside networks.

7.1.3 Peer discovery

Peers discover other peers either through a centralized tracker or through local multicast announcements. Peers communicate with the tracker through a request-reply protocol, where they announce their membership state and in return receive a list of peers to connect with. Peers also send periodic announcements as link-local multicast messages to their direct neighbours, allowing those neighbours to discover them and initiate a connection.

This hybrid approach to peer discovery makes the most use of local clusters of peers, while also preventing partitions to be created where some peers are unable to receive the full file. This allows the solution to work even in highly heterogeneous WSN deployments, which is not well supported by other solutions.

7.1.4 File distribution

Peers distribute the file through a P2P protocol, where every peer helps in distributing (parts of) the entire file to other peers.

Peers try to connect with other peers discovered through both discovery mechanisms. They send periodic announcements to all connected peers, notifying them of their download progress. Peers are made robust against failures and lost transmissions through the use of retry counters and timers.

Like BitTorrent, the distributed file is subdivided into pieces which can be independently distributed among peers, allowing peers to download multiple pieces in parallel to speed up distribution. Pieces are exchanged through a request-reply protocol, where peers request a missing piece that is available at a connected peer. The piece exchange also takes advantage of local clusters, by sending piece data to all local peers using a multicast when one local peer requests the piece.

The protocol runs on top of IPv6, allowing it to work across multiple networks and even across the Internet. For example, the tracker can be located both inside the WSN or in a separate network, and a file could even be distributed across multiple WSNs simultaneously. This flexibility enables support for larger and more heterogeneous network configurations, ranging from small-scale multi-lab set-ups to city-wide multi-network applications.

7.1.5 Implementation

A prototype was implemented for both the tracker and the peer. The prototype is rather minimal, but allows for a decent initial evaluation of the proposed solution.

The tracker is implemented as a Java command-line application, to run in an network external to the WSN. It tracks the swarm of all known torrents, responds to incoming announce requests and provides requesting peers with a list of other peers to connect with.

The peer is implemented as a Contiki application in C, to run on nodes inside the WSN. It is designed with the resource constraints of these nodes in mind. Some extra tools were developed to help in setting up a file distribution using NanoTorrent.

7.1.6 Evaluation

The prototype implementation has been evaluated in the COOJA simulator. The scalability was tested by running the prototype in simulated networks of increasing size. The proposed use case of a heterogeneous network consisting of different types of nodes was also evaluated in a network set up with two separated node clusters.

The results show that NanoTorrent's hybrid peer discovery comes with a trade-off. The hybrid approach allows for faster distribution, but uses a lot of transmissions on the network. Whereas other distribution protocols such as Deluge [16] only discover local neighbours, NanoTorrent can find distant nodes which are multiple hops away by fully using the potential of IPv6. This allows it to work in more heterogeneous networks, but makes it more inefficient when used in homogeneous WSN deployments.

7.2 Lessons learned

NanoTorrent explores an opportunity to combine two peer discovery mechanisms in an attempt to take the best from both worlds. It turns out that there's a trade-off to be made between faster distribution speed and reduced communications. This is of course to be expected, as there's no such thing as a 'free lunch' when it comes to real-life performance. There is still room for improvement in terms of transmissions, but the overhead from maintaining two parallel peer discovery mechanisms is unlikely to disappear.

The choice to rely solely on IPv6 for the protocol's design gives NanoTorrent an advantage over other protocols by making multi-hop and cross-network operation possible by default. As WSNs scale up to larger dimensions and vast deployments, designers of WSNs protocols should consider the opportunity to rely on the available IPv6 infrastructure, rather than re-inventing the wheel with custom routing protocols. More investigation and effort is needed to make 6LoWPAN more feature-rich (e.g. security), but it is already a very good candidate as basis for a protocol.

7.3 Future work

7.3.1 Parameter tuning

As mentioned in section 6.5, many of the protocol's configuration parameters are not yet fully fine-tuned and their impact on the performance is not yet explored. Addi-

tional experiments could be carried out to test different combinations of parameters, and to optimize them for specific use cases.

7.3.2 Large-scale experiments

The scalability experiments from 6.3 stress-test the protocol in networks of up to 50 nodes. While this is already a relatively large network, it comes nowhere near the wide-scale deployments of hundreds or thousands of nodes envisioned for IoT applications. Experiments with more nodes in different network topologies are needed to further investigate the scalability of the protocol.

7.3.3 Transmission reductions

The results from chapter 6 indicate that there is still a lot of effort needed to make NanoTorrent efficient with transmissions. Section 6.5 discusses some possible improvements, such as removing duplicate peer connections, pipelining piece data replies or sending more data in a single data reply. These improvements are yet to be explored.

7.3.4 Comparison with other protocols

Given the variety of existing file distribution protocols for WSNs, a comparison of NanoTorrent versus one of these other protocols could prove valuable in learning more about their performance in various use cases. Although a comparison with Deluge [16] was originally planned, it could not be completed due to technical difficulties and time constraints.

Appendices

Appendix A

Popular article

BitTorrent on a diet

Peer-to-peer file distribution in wireless sensor networks

Mattias Buelens, *KU Leuven*

The internet is in everything. Just a few decades ago, you had to sit in front of a big clunky machine on your desktop and dial in through a sluggish 56k modem to connect to the internet. Today, you can surf the web on your laptop, your mobile phone, your TV and even your wristwatch. And the invasion of the internet doesn't stop there. In the future, the "internet of things" will connect many small battery-powered devices to work together and help monitor and control an environment.

However, how do you keep the software on these devices up-to-date without wasting too much battery and reducing their lifespan? Mattias Buelens, an engineering master student from KU Leuven university, is working on a thesis to design an implement a protocol which can serve such software updates fast and efficiently by taking some ideas from the peer-to-peer file transfer protocol BitTorrent.

Context

In the "internet of things", everything from your washing machine to your central heating system will have an internet connection so you can monitor and control them from everywhere. These small embedded devices communicate with you and with each other in a so-called "wireless sensor network" (WSN) to measure and react upon their environment. They can be tasked with anything from simply keeping your house warm during the winter, to monitoring an active volcano and alerting for possible eruptions as shown in figure 1.

These embedded devices are installed with a sufficient battery and a wireless connection to do their job for many years without interruption. Often, it is impractical or impossible to manually fix such a device after its installation. For example, these devices may be embedded in a thick wall, or may be dropped near the crater of a volcano. When a bug in the initial software is discovered or a new feature is developed, it is necessary that the updated software can be deployed without physical access to the device. Using their wireless connection, these devices can receive an 'over-the-air' update to their configuration or software to keep them running for

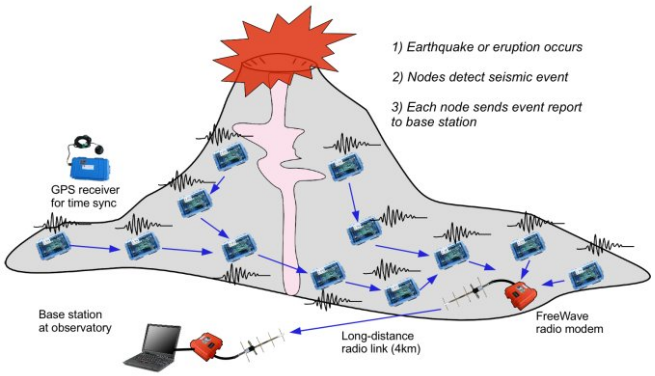


Figure 1: *A wireless sensor network deployed by the Harvard Sensor Networks Lab on an active volcano to monitor seismic activity. [1]*

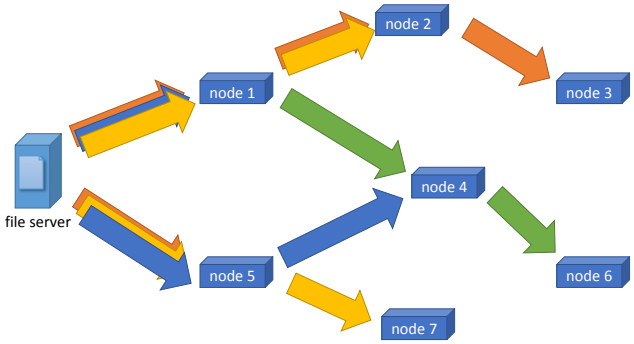


Figure 2: *Illustration of network traffic generated by a naive file distribution protocol. The file server needs to send the complete file over different separate connections to every node in the network. This leads to a lot of redundant traffic passing through the nodes closer to the server.*

many more years to come.

However, such wireless software deployments eat up quite a bit of battery juice, which can reduce the lifespan of the device. When all devices naively start to download a new software version from one single source, the nodes closest to the source will have to route a lot of data from the source to many distant nodes in the network. This large amount of network traffic going through a few nodes causes these nodes to use up more energy and therefore reduce their lifespan more quickly than more distant nodes. This load imbalance is bad for the whole network: if the nodes close to the source fail sooner, the rest of the network could become disconnected from the source. Moreover, most of the generated traffic is redundant, since the same file passes multiple times over some nodes in the network to be routed to distant nodes (figure 2).

Therefore, a smarter deployment protocol is needed which better balances the load over all devices, utilizes the knowledge about the file at other nodes to reduce redundant traffic, but still deploys the software reasonably fast to every node in the network.

Inspiration from BitTorrent

What is BitTorrent?

Many people have already heard about or even used BitTorrent to download large files on the internet, such as movies, games or complete Linux distributions. However, few know what it actually is or how it works.

In essence, BitTorrent is a peer-to-peer file transfer protocol: instead of downloading a file from one single source (server), you download pieces of the file from other people who already downloaded that piece for themselves. Your computer becomes a 'peer' in the torrent's 'swarm', and starts downloading pieces from other peers but also uploading your downloaded pieces to others.

This allows others to benefit from your upload capacity, which would otherwise be unused by a standard client-server protocol such as FTP. This is one of the original design goals of BitTorrent, and manifests itself clearly in the internet statistics where it is responsible for 25% of the world's upload traffic [2].

In order to join the swarm, BitTorrent uses a centralized tracker which is responsible for keeping track of the swarm. A new peer first contacts the tracker, which replies with a list of peers to connect with. From then on, the peer can connect with (some of) these other peers and start downloading pieces. When a peer wants to leave the swarm, it also announces this to the tracker so it will no longer advertise it to other peers. Although the tracker is a centralized entity and thus a potential single point of failure, the file transfer itself is fully decentralized. This means that the tracker only needs to store the torrent's information and the list of all peers in the swarm,

and doesn't need to know the actual contents of the file being distributed.

This is especially interesting for content providers who want to serve large files to many clients. Instead of putting the full load of the file distribution on a single server, BitTorrent can share the load over every peer in the swarm. As long as every piece of the file is available from at least one peer in the swarm, every peer can receive the whole file. A content provider therefore doesn't need to install a very powerful server to handle thousands of concurrent downloads: it only needs to keep one server in the swarm as the 'initial seed' with the whole file, and the load is shared as soon as other peers download some pieces from the seed. When *more* peers are downloading the file, the load on the initial seed actually *decreases* as opposed to increasing in the case of classical server-client protocols.

Why BitTorrent in a WSN?

Although BitTorrent was designed for file distribution across the whole internet, its ideas can be very useful for solving design challenges for our protocol.

BitTorrent provides load balancing over the peers in the swarm, which is especially interesting for wireless sensor networks. Instead of having every node download from a single server and generating a lot of traffic for the nodes near that server, the traffic can be more scattered over the whole network. Ideally, every node downloads and uploads about the same amount, draining every node's battery equally.

It is also fairly resilient to individual failures. When a peer leaves the swarm or crashes, the other peers can still communicate and continue downloading. When the peer rejoins the swarm later, it knows which pieces it had already downloaded and resume downloading the remaining pieces. This is useful for BitTorrent when downloading very large files in multiple sessions over many hours or even days, where having to restart the download after an interruption can mean a lot of wasted time. For a wireless sensor network deployed in a harsh environment, this kind of fault tolerance and recovery can allow for longer lifetime of the network.

Optimizations for WSNs

Although BitTorrent is a good reference to start designing a peer-to-peer file transfer protocol, it cannot be used as-is for use in a wireless sensor network. WSNs are structured differently than the internet, nodes in a WSN are not as powerful as consumer computers, and battery consumption needs to be taken into account. This leads to optimizations in the protocol design targeted for WSNs.

Simple tracker for pairing up nodes

The designed protocol uses a tracker much like BitTorrent's version, although in a slimmed down form. BitTorrent uses HTTP for its tracker communication and adds a lot of metadata for statistical purposes to the tracker messages (such as how much the peer has already uploaded or downloaded), which is not needed for the purposes of a wireless sensor network. Instead, the tracker communicates over UDP in small, structured messages and only keeps track of the list of peers in the swarm of a torrent.

Smaller files and pieces

BitTorrent is used mostly for distributing very large files, often many GB in size. It splits up the file in fairly large pieces, usually a couple of MB per piece. A wireless sensor node only has a couple of KB of space, so the maximum file size and piece size for the designed protocol is very different. The protocol is designed for distributing small files around 2 to 4 KB in size, such as executable programs, configuration files or stored sensor measurements. These files can then be split up and distributed in pieces of around 256 bytes.

Local peers for reduced routing

In order to achieve high download speed, it is important that you select peers which can provide a high upload speed *to you*. This doesn't mean that everyone will get a high upload speed from a peer, the actual speed also depends on how close you are to that peer. A computer in Japan might have an upload bandwidth of 10

MB/s, but if the network between you and that Japanese peer is poor, you might only get 100 KB/s download speed on your end. On the other hand, your neighbour's computer uploading at 1 MB/s can probably provide you with a much better download speed, although it won't be able to serve other Japanese peers very well.

BitTorrent clients achieve this by monitoring the upload and download speed of each peer they're connected with, and then downloading from the peer which provides the best upload speed for you. This involves a lot of bookkeeping, and needs carefully considered strategies to work. These strategies also take into account punishing peers when they download a lot but do not upload in response ('leeching'), in order to discourage this unfair behaviour.

For wireless sensor networks, fairness is less of an issue: we assume that all nodes run the same protocol implementation and will not attempt to leech from others. Therefore, a simpler mechanism is employed: peers select a number of neighbouring peers, and also a few (two or three, depending on network size) remote peers which are farther away. They should then be able to download most of the file from their neighbours at high speeds, and receive missing pieces from the remote peers. In the worst case, a piece may be missing on both the neighbouring peers as well as the remote peers, in which case the peer needs to connect with a different remote peer (and optionally contact the tracker). This mechanism should allow for less negotiation overhead, while still providing decent piece availability and fairly high download speeds.

Future work

Although the protocol design is sufficiently complete for a first implementation, it is far from being finalized.

For example, the protocol does not yet take the battery consumption of a node into account. It is possible that two peers both have a particular piece available, but their neighbouring peers all pick the same peer to download the piece from. This means that one peer has to send a lot of data, while the other is left underused. One possible solution would be to let peers keep track of their uploaded and downloaded amounts, and regularly announce these amounts with their neighbouring peers. The neighbours could then use this information to make a better selection of the peer to download from, and pick a less strained peer over an overworked one.

Experiments will be performed to analyze the impact on battery consumption to find out what improvements are worth implementing in the protocol.

Conclusion

Wireless sensor nodes in the "internet of things" need their software updates delivered fast but energy efficient. Therefore, a new protocol is being designed to distribute files on such nodes in a peer-to-peer fashion. The protocol is inspired by the popular BitTorrent file transfer protocol, but optimizes for wireless sensor networks by using smaller file and piece sizes and by relying on locality for high download speeds. With the major design decisions for the protocol complete, the protocol will now be further developed, implemented and tested on real wireless sensor nodes.

The major benchmark for the protocol will be if it indeed performs better than standard client-server file transfers, and in what circumstances. Intuitively, a client-server approach will have less overhead in a very small network where all nodes can directly reach the server. For larger networks, the peer-to-peer approach should theoretically perform better, but it will need more communication for bookkeeping purposes. Experiments will have to decide how much overhead is created in various configurations and if this indeed improves over the client-server approach.

In June 2015, you'll hopefully get an answer from Mattias Buelens when he presents his master's thesis.

References

[1] Harvard Sensor Networks Lab. Volcano monitoring. http://fiji.eecs.harvard.edu/Volcano, 2008.

[2] Sandvine. Global Internet Phenomena Report: 1H 2014. https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf, 2014.

Appendix B

IEEE article (Dutch)

NanoTorrent: Plaatsbewuste peer-to-peer bestandsdistributie in draadloze sensornetwerken

Mattias Buelens, *KU Leuven*

***Samenvatting*—Draadloze sensornetwerken bestaan uit vele kleine computers uitgerust met sensoren en actuatoren die met elkaar kunnen communiceren dankzij hun draadloze antenna. Om deze netwerken operationeel te houden voor lange tijdspannes, moeten ze zich kunnen aanpassen en evolueren ook nadat ze uitgezet zijn in hun omgeving. Dit betekent dat ze grote bestanden zoals nieuwe programma's of configuraties snel en efficiënt via het netwerk moeten kunnen verkrijgen.**

NanoTorrent is een peer-to-peer protocol dat snelle bestandsdistributie toelaat in draadloze sensornetwerken. Het introduceert een hybride mechanisme om peers te vinden, door zowel met een gecentraliseerde tracker te werken en in de lokale omgeving naar buren te zoeken. Dit laat toe om zelfs in zeer heterogene netwerken te opereren, waarbij verschillende knopen verschillende programma's uitvoeren. De peer-to-peer aanpak helpt om de belasting door de bestandsdistributie te verdelen over het netwerk, en maakt gebruik van link-local multicast berichten om delen van het bestand tegelijkertijd naar meerdere buren te distribueren.

De evaluatie van het protocol toont aan dat NanoTorrent een snelle bestandsoverdracht kan realiseren in verschillende netwerkconfiguraties. De hybride aanpak om peers te vinden komt wel met een trade-off waarbij de toegenomen overdrachtssnelheid ook extra transmissies met zich meebrengt. Deze berichten moeten in meerdere hops over het netwerk om afgelegen knopen te bereiken, waardoor de tussenliggende knopen moeten helpen om deze door te sturen. Het protocol kan ook op nog verschillende plaatsen verbeterd worden om minder verkeer over het netwerk te sturen.

***Keywords*—Draadloze sensor netwerken, internet der dingen, peer-to-peer, bestandsdistributie, draadloze netwerken**

I. Inleiding

DRAADLOZE sensornetwerken zijn een opkomende vorm van computernetwerken bestaande uit vele kleine computers met beperkte mogelijkheden en capaciteiten die uitgerust zijn met allerlei sensoren en actuatoren en met elkaar draadloos kunnen communiceren. Deze computers of 'knopen' werken samen om metingen te verrichten, gegevens te verzamelen en te verwerken en acties te plannen en uit te voeren. Draadloze sensornetwerken kunnen uitgezet worden in voor mensen moeilijk bereikbare of gevaarlijke omgevingen, zoals in metrotunnels of op vulkanen [1]. Hier kunnen ze wetenschappelijke metingen doen, gevaren in de omgeving detecteren of zelf beslissen om actie te ondernemen.

Omdat deze netwerken voor lange tijd moeten meegaan en mogelijk niet meer fysiek toegankelijk zijn nadat ze zijn geïnstalleerd, moeten ze kunnen aangepast worden op afstand. Nieuwe toepassingen of nieuwe taakconfiguraties moeten kunnen verspreid worden naar alle knopen in het netwerk, en deze moeten zich zelfstandig kunnen herconfigureren met de nieuwe data. Deze herprogrammering of herconfiguratie van het netwerk moet liefst zo snel mogelijk gebeuren, zodat de knopen kunnen verder gaan met hun eigenlijke taak. Omdat de kleine computers vaak op batterijen werken, moet dit ook energie-efficiënt gebeuren met zo weinig mogelijk uitgewisselde berichten tussen knopen.

NanoTorrent is een peer-to-peer protocol dat zulke grote bestanden kan verspreiden over een draadloos sensor netwerk. Hiervoor gebruikt het een hybride mechanisme om peers te vinden, door zowel met een gecentraliseerde tracker te werken en in de lokale omgeving naar buren te zoeken. De peer-to-peer bestandsoverdracht helpt om de belasting van de gecommuniceerde berichten te verdelen over het hele netwerk.

Het vervolg van dit artikel is als volgt gestructureerd. Sectie II geeft een overzicht van de probleemcontext van internet der dingen en draadloze sensornetwerken. Sectie III bespreekt het ontwerp van het NanoTorrent protocol. Sectie IV beschouwt de implementatie van het gemaakte prototype. Sectie V evalueert het ontwerp en het prototype. Sectie VI bespreekt het onderzochte gerelateerde werk. Sectie VII eindigt het artikel met de conclusies.

II. Context

A. Internet der dingen

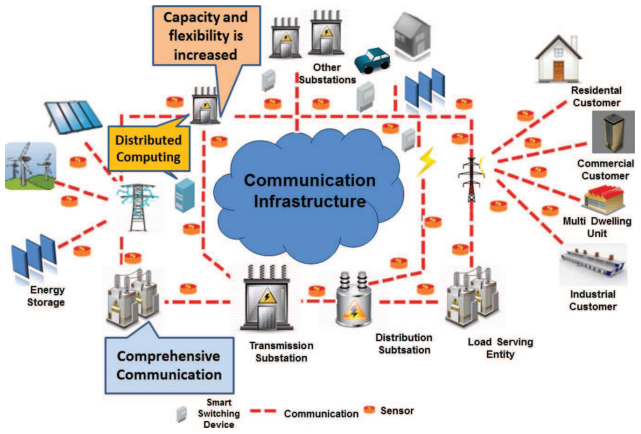
In de visie van het 'internet der dingen' worden fysieke objecten uitgerust met kleine ingebouwde computers die het mogelijk maken om deze dingen te monitoren en te besturen op elk moment van eender waar ter wereld. Dankzij IPv6 kan ieder object een uniek adres toegewezen worden dat via het internet bereikbaar is [2].

De mogelijke toepassingen van het internet der dingen beslaan bijna alle domeinen, gaande van het verbeteren van het comfort thuis tot het controleren van grote essentiële infrastructuren:

Slimme huizen. Door huishoudtoestellen, temperatuursensoren en beveiligingscamera's te verbinden met het internet, kan een huiseigenaar op ieder moment zijn leefcomfort controleren met elk toestel [3]. Met een enkel dashboard krijgt hij een overzicht van alle toestellen thuis, en kan hij bijvoorbeeld zowel de airconditioning als de rolluiken aansturen.

Slimme elektriciteitsnetten. Met de verschuiving van fossiele brandstoffen naar hernieuwbare energiebronnen wordt energieproductie steeds minder voorspelbaar. Terwijl kool- en kerncentrales elektriciteit kunnen produceren op ieder moment, hangt de productie van zonnepanelen en windmolenparken af

M. Buelens is een masterstudent aan de KU Leuven, Leuven, België.



Figuur 1. Voorbeeld van een architectuur voor een slim elektriciteitsnet. [5]

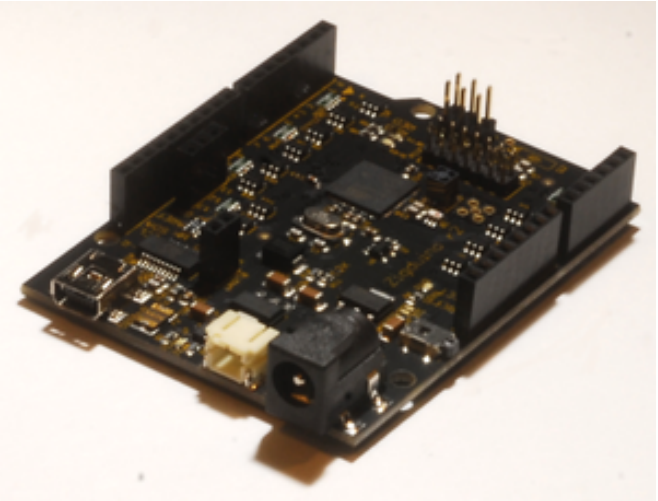
van de tijd en de weercondities. Dit kan leiden tot tekorten of blackouts, of overproductie. Om te kunnen omgaan met deze fluctuaties, moeten slimme elektriciteitsnetten de mogelijkheid hebben om het energieverbruik nauwkeuriger te voorspellen op korte termijn, en om verbruikspieken te verschuiven naar momenten waarop er meer productiecapaciteit is [4]. Huizen kunnen uitgerust worden met slimme elektriciteitsmeters die continue het verbruik meten en doorsturen naar de producenten, en wasmachines kunnen geprogrammeerd worden om te werken wanneer er weinig verbruik op het net is. Dit biedt veel uitdagingen om een gesofisticeerde IT infrastructuur te ontwikkelen die deze slimme apparaten kan ondersteunen [5], zoals getoond in figuur 1.

B. Draadloze sensornetwerken

Vele toepassingen in het internet der dingen worden geïmplementeerd met een draadloos sensornetwerk. Deze bestaan uit een aantal kleine computers (ook wel ‘knopen’ of ‘motes’ genoemd) met sensoren, actuatoren en een draadloze antenne. De knopen meten veranderingen in hun omgeving, communiceren met elkaar en de buitenwereld en werken in op hun omgeving.

Knopen in een draadloos sensornetwerken verschillen sterk van gewone computers, en hebben verschillende ontwerpvereisten voor hun programma’s:

- Door hun kleine grootte zijn ze beperkt in hun mogelijkheden. Terwijl gewone computers multi-core gigahertz processors hebben met RAM geheugen in de orde van GB, hebben knopen slechts megahertz microprocessoren en een paar KB aan RAM. De TMote Sky heeft bijvoorbeeld een kloksnelheid van 8 MHz en 48 KB aan geheugen [6], terwijl de AVR Zigduino r2 (figuur 2) aan 16 MHz draait met 128 KB aan flash RAM [7].
- De levensduur van een knoop is beperkt door zijn batterij. Dit maakt energieverbruik een belangrijke ontwerpvereiste voor programmeurs.
- Draadloze sensor netwerken zijn onbetrouwbaar. Knopen kunnen falen doordat hun batterij opgebruikt zijn of doordat de omgeving hen fysiek vernielt (overstromingen, branden, aardbevingen,...) Programmeurs moeten rekening houden met falende knopen.
- Draadloze netwerken zijn erg gedistribueerd. Een netwerk kan bestaan uit honderden knopen, met mobiele knopen die binnen en buiten het bereik van de antenne vallen. Gedistribueerde programma’s moeten kunnen schalen naar zulke grote aantallen van knopen.



Figuur 2. Het AVR Zigduino r2 platform. [7]

C. BitTorrent

Het voorgestelde protocol haalt veel inspirate van het BitTorrent protocol [8]. Dit is een peer-to-peer bestandsdistributieprotocol voor het internet, dat het bestand opsplitst in gelijke delen (‘pieces’) die afzonderlijk verspreid kunnen worden tussen peers. Gebruikers moeten eerst een `.torrent` bestand downloaden om de eigenlijke distributie van de ‘torrent’ te starten in hun BitTorrent client.

Peers melden zich aan bij een centrale tracker, die hun deelname registreert in de ‘swarm’ en een lijst van andere peers terug geeft. Peers kunnen vervolgens verbinding maken met de ontdekte peers, en ontdekken welke pieces de andere peers al hebben gedownload. Een peer kan dan een piece dat hij ontbreekt aanvragen bij een andere peer, het piece ontvangen en vervolgens zelf verspreiden naar andere peers.

BitTorrent zorgt ervoor dat peers ook moeten helpen met pieces te uploaden, en niet enkel mogen downloaden. Hiervoor gebruikt het een tit-for-tat strategie, waarbij peers die weigeren om pieces aan te bieden gestrafd worden doordat andere peers ook zullen weigeren om pieces aan te bieden aan deze slechte peer. Dit maakt dat alle peers verplicht worden te helpen bij de distributie, en zo de belasting over de verschillende peers eerlijker verdeeld wordt.

NanoTorrent neemt de concepten van torrents, trackers, swarms en pieces over in het ontwerp. Peers worden op een gelijkaardige manier gevonden via een tracker, en bestanden worden als pieces verspreid over het peer-to-peer netwerk.

III. Ontwerp

Het ontwerp van het NanoTorrent protocol bestaat uit twee grote delen: het vinden van peers om mee te verbinden, en het uitwisselen van het bestand met deze gevonden peers.

A. Vinden van peers

NanoTorrent gebruikt twee parallele mechanismen om peers te vinden.

Peers vinden met tracker. Een eerste manier bestaat erin om peers te laten communiceren met een centrale 'tracker', die bijhoudt welke peers er momenteel deelnemen aan de distributie van de torrent. Wanneer de tracker een aanvraag ontvangt van een peer, voegt hij deze peer toe aan de swarm en geeft als antwoord een aantal andere peers uit dezelfde swarm terug. De peer kan dan verbindingen beginnen maken met peers uit de ontvangen lijst. Opdat de tracker geen peers teruggeeft die al lang niet meer reageren en dus geen verbindingen zullen accepteren van peers, moeten peers periodiek hun deelname bij de tracker vernieuwen. De tracker kan dan oude peers na een tijd verwijderen, zodat deze niet meer voorkomen in de antwoorden.

Lokale peers. Een tweede manier gebruikt de nabijheid van andere peers om deze te vinden. Peers sturen periodiek een bericht naar alle buren in hun omgeving met informatie over de torrent die zij momenteel aan het downloaden zijn. Hiervoor maken zij gebruik van IPv6 link-local multicast berichten. Wanneer een peer zo'n multicast bericht ontvangt, kan hij zien van welke buur dit bericht kwam en zelf een verbinding beginnen met deze buur. Doordat berichten over een draadloos medium hoe dan ook hoorbaar moeten zijn voor alle omringende buren, kost het verzenden van zo'n multicast bericht even veel als een gewoon unicast bericht gericht naar slechts één ontvanger.

Door deze twee mechanismen samen te gebruiken, kan het protocol meer heterogene draadloze netwerken ondersteunen bestaande uit een aantal clusters van knopen. Binnen een cluster kunnen knopen elkaar vinden met lokale berichten, terwijl ze knopen uit andere clusters kunnen vinden via het antwoord van de tracker.

B. Bestandsdistributie

De peer-to-peer bestandsdistributie is erg gelijkaardig aan dat van BitTorrent. Het heeft wel enkele belangrijke aanpassingen met het oog op de beperkte capaciteiten van knopen in een draadloos sensornetwerk.

Peers proberen verbindingen op te zetten met alle peers die ze weten te vinden. Voor elke verbinding moeten ze een beetje geheugen reserveren om de toestand bij te houden. Deze toestand bestaat uit: het IPv6 adres van de andere peer, de set van pieces die beschikbaar zijn bij de andere peer, het tijdstip van het laatste bericht ontvangen van deze peer, en informatie over de huidige aanvraag voor een piece die onlangs verstuurd is naar de peer. Peers onderhouden hun verbindingen door periodiek `have` berichten te versturen, waarin ze hun huidige set van beschikbare pieces meedelen aan verbonden peers. De ontvangende peers werken dan hun beeld op de toestand van die peer bij.

Wanneer een peer merkt dat een ontbrekende piece beschikbaar is geworden bij een van zijn verbonden peers, kan deze een aanvraag versturen voor die piece. De peer verstuurt als antwoord de inhoud van die piece, zodat de ontvanger deze data kan wegschrijven en zelf de piece kan aanbieden aan zijn eigen connecties.

Als optimalizatie kunnen peers die een aanvraag voor een piece ontvangen van een lokale peer, hun antwoord met de data versturen naar *alle* buren als een multicast bericht. Dit laat lokale peers toe om data te ontvangen voor pieces die zij nog niet expliciet hadden aangevraagd, maar wel in geïntereseerd zijn. Zo kan een lokale piece uitwisseling meerdere lokale peers tegelijk bedienen.

IV. Implementatie

De systeemarchitectuur van de implementatie van het prototype is weergegeven in figuur 3. Hierin is de tracker verantwoordelijk voor het beheer van de swarm, de peer verantwoordelijk voor het vinden van andere peers en het uitwisselen van pieces, en de border router verantwoordelijk voor de verbinding tussen het draadloze sensornetwerk en het externe netwerk waar de tracker zich bevindt.

A. Tracker

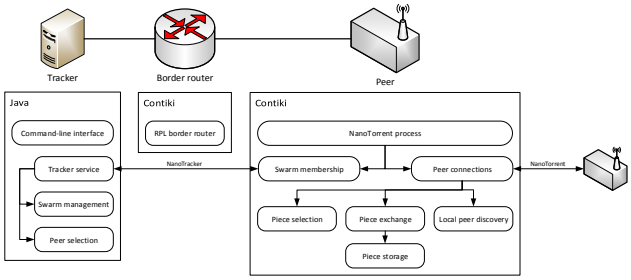
De tracker is geïmplementeerd als een stand-alone Java applicatie. Peers communiceren met de tracker over het NanoTracker UDP protocol om deel te nemen aan een swarm en peers te vinden in de swarm. De tracker onderhoudt de toestand van alle swarms en alle peers die hij opvolgt, en gebruikt deze informatie om een lijst van andere peers in de swarm aan te bieden aan nieuwe peers. In de huidige versie van het prototype kiest de tracker gewoon een aantal peers uit de swarm willekeurig, zonder een intelligent selectie algoritme. Dit kan indien nodig aangepast worden om meer informatie in rekening te houden bij deze keuze, zodat peers betere verbindingen kunnen opzetten.

B. Peer

De peer is geïmplementeerd in C als een Contiki [9] applicatie. Deze moet o.a. de verbinding met de tracker onderhouden, verbindingen met peers maken en accepteren, pieces kiezen om aan te vragen bij andere peers en op zijn beurt pieces aanbieden aan andere peers.

Om te vermijden dat peers te veel connecties tegelijkertijd zouden openen, zijn het aantal tegelijk geopende connecties beperkt. Wanneer een peer een nieuwe connectie probeert te openen, kan het dus zijn dat dit mislukt. De peer stuurt een `close` message sturen naar de andere peer om te laten weten dat hij de connectie niet kon openen, zodat deze aan zijn kant de eigen toestand ook kan opkuisen.

Peers gebruiken net zoals BitTorrent een *zeldzaamste eerst* policy om pieces te selecteren voor aanvragen. Hierbij verkiest de peer om eerst de piece aan te vragen die het minst voorkomen bij zijn verbonden peers. Dit helpt om deze zeldzame pieces sneller te verspreiden over de rest van het netwerk,



Figuur 3. NanoTorrent systeemarchitectuur

zodat ze sneller minder zeldzaam worden en de belasting op het netwerk weer beter verdeeld kan worden.

Peers zorgen er ook voor dat ze eenzelfde piece niet bij meerdere peers tegelijk aanvragen. Dit is inefficiënt, omdat de peer beter verschillende pieces in parallel kan aanvragen. Een uitzondering op deze regel is wanneer de peer bijna alle pieces heeft gedownload. In deze *end-game modus* tracht de peer zo snel mogelijk de paar ontbrekende pieces te downloaden met meerdere aanvragen bij verschillende peers, zodat de peer sneller het volledige bestand heeft en andere peers optimaal kan helpen met hun distributie.

Pieces worden opgeslagen in het Contiki File System (CFS). Afhankelijk van het platform waarvoor de Contiki applicatie wordt gecompileerd, kan dit belanden op een harde schijf, een EEPROM geheugen of in RAM geheugen. Op het AVR Zigduino platform kan CFS naar 4 KB aan EEPROM geheugen schrijven [7]. CFS maakt abstractie van deze verschillende opslagmedia, en biedt een uniforme interface om bestanden te lezen en te schrijven.

V. Evaluatie

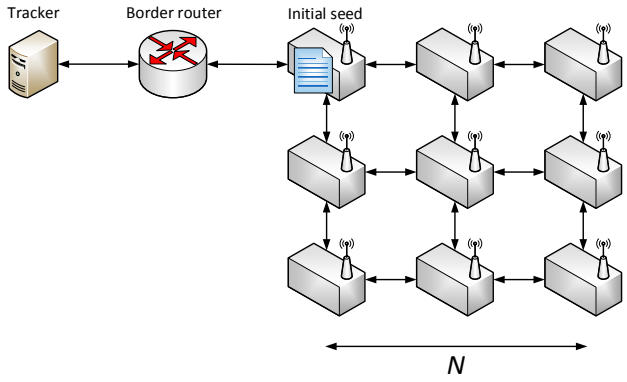
Het prototype werd geëvalueerd op zijn schaalbaarheid en op zijn bruikbaarheid in een heterogene netwerkconfiguratie door middel van de COOJA simulator.

Het doel van NanoTorrent is om bestanden te distribueren naar vele knopen in een draadloos netwerk, in een aanvaardbare tijd en met zo weinig mogelijk berichten om energie-efficiënt te zijn op fysieke knopen. Het moet bruikbaar zijn in verschillende netwerkconfiguraties bestaande uit verschillende soorten knopen met verschillende programma's, die mogelijk niet allemaal deelnemen aan de NanoTorrent distributie.

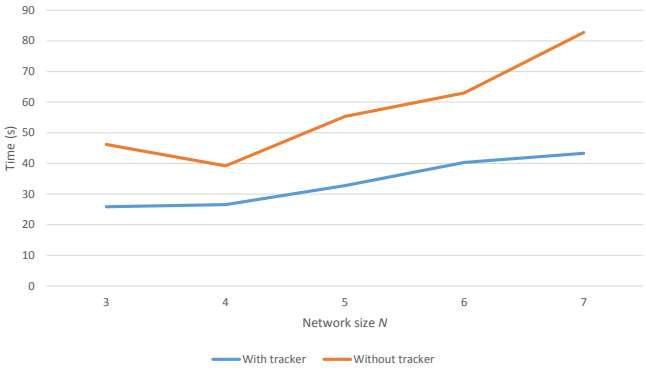
COOJA [10] maakt het mogelijk om een volledig Contiki platform te simuleren zonder wijzigingen aan de code van de geteste applicatie. In de simulator kan eender welke configuratie van knopen getest worden, met verschillende parameters voor het communicatiemedium. Tijdens de simulatie kan alle uitvoer en verzonden pakketten weergegeven en opgeslagen worden, wat zeer nuttig is om later analyses te maken.

A. Schaalbaarheid

De schaalbaarheid van het protocol werd getest door een bestand te distribueren in een netwerk met N^2 knopen in een $N \times N$ rasterconfiguratie, zoals in figuur 4. De resultaten zijn getoond in figuur 5 en 6.



Figuur 4. Netwerkconfiguratie voor schaalbaarheidsexperiment. Het netwerk bestaat uit $N \times N$ knopen uitgelijnd in een raster, met één initiële seed.



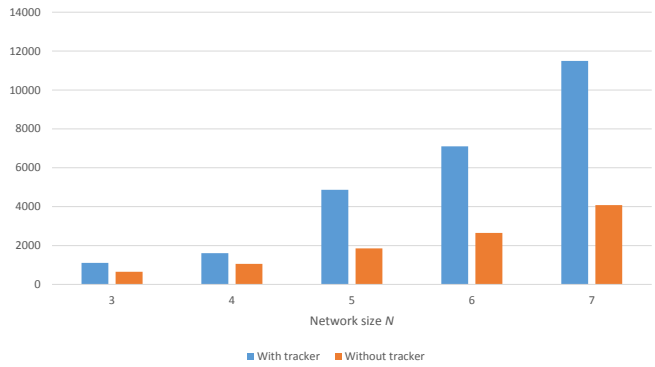
Figuur 5. Tijd nodig om het volledige bestand te distribueren naar alle knopen voor een netwerkdiameter N.

Het gebruik van een tracker zorgt duidelijk voor een snellere distributie, doordat peers meer verbindingen kunnen maken en zo sneller pieces kunnen aanvragen. Zonder een tracker moeten peers wachten totdat hun onmiddellijke buren pieces beginnen te ontvangen.

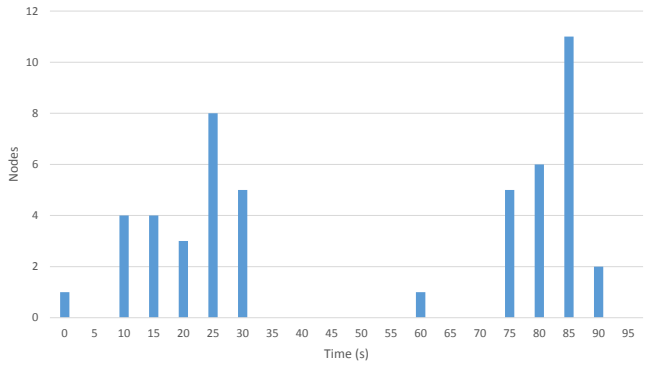
Echter, de toegenomen distributiesnelheid zorgt voor een drastische toename in het aantal verzonden berichten. Omdat peers communiceren met ver afgelegen peers ontdekte via de tracker, moeten hun berichten langs meerdere tussenliggende knopen passeren vooraleer ze hun bestemming bereiken. Deze tussenliggende knopen moeten hun eigen energie gebruiken om deze extra berichten door te sturen naar de juiste bestemming, wat zorgt voor meer verkeer op het netwerk. Wanneer de tracker niet gebruikt wordt, blijft het aantal verzonden berichten veel meer beperkt en schaalt het beter naarmate het netwerk groter wordt.

B. Heterogeniteit

Terwijl andere protocols zoals Deluge [11] vereisen dat alle knopen in het netwerk hetzelfde protocol draaien, kan Nano-



Figuur 6. Total aantal transmissies verzonden tijdens de distributie voor een toenemende netwerkdiameter N.



Figuur 7. Voltooiingstijden in netwerk met twee 5×5 clusters.

Torrent opereren in heterogene netwerken waarbij sommige knopen geen NanoTorrent implementatie uitvoeren.

Om deze toepassing te valideren, werd een experiment uitgevoerd met twee clusters van 5×5 knopen die enkel verbonden worden door een border router. De initiële seed werd geplaatst in één van de twee clusters, zodat peers in de andere cluster moeten verbindingen maken met deze cluster om het bestand te kunnen verkrijgen.

De verdeling van de voltooiingstijden van de verschillende peers is getoond in figuur 7. Hieruit blijkt dat het tot een minuut duurt vooraleer de eerste knoop in de andere cluster het volledige bestand heeft ontvangen. Zodra de eerste knoop in deze cluster de download heeft voltooid, kan het bestand wel sneller verspreiden doorheen de cluster, zodat alle knopen na in totaal anderhalve minuut het volledige bestand hebben.

C. Mogelijke verbeteringen

Er is duidelijk nog ruimte voor verbetering aan het protocol. Zeker het aantal verzonden berichten bij het gebruik van een tracker moet nog efficiënter gemaakt worden om bruikbaar te zijn in echte draadloze netwerken.

De configuratieparameters van het protocol moeten nog beter fijngesteld worden. Deze bepalen o.a. de waarden van de timers en het aantal tegelijk geopende connecties, die een grote impact kunnen hebben op de performantie. Periodieke `have` berichten kunnen initieel sneller verstuurd worden om sneller te kunnen beginnen met pieces te verspreiden. Momenteel is er ook nog weinig coördinate tussen knopen bij het versturen van aanvragen voor pieces, in vergelijking met Deluge [11] dat redundante aanvragen zoveel mogelijk tracht te vermijden.

Het is ook nog mogelijk dat peers twee verbindingen maken met dezelfde peer. Peers kunnen niet achterhalen of een peer gevonden via een tracker en via lokale multicast aankondigingen dezelfde fysieke node draaien.

D. Discussie

Uit de resultaten van het schaalbaarheidsexperiment volgt dat het gebruik van een tracker een trade-off met zich meebrengt. De distributiesnelheid kan verhoogd worden, ten koste van meer berichten over het netwerk.

NanoTorrent laat echter wel toe om in meer netwerkconfiguraties bestanden te distribueren doordat het IPv6 gebruikt als onderliggend netwerkprotocol. Het heterogeniteitsexperiment toont aan dat NanoTorrent een bestand kan distribueren tussen twee clusters gescheiden door een IPv6 border router, iets waar andere protocollen niet in slagen.

VI. Gerelateerd werk

A. Local Peer Discovery for BitTorrent

Local Peer Discovery [12] is een uitbreiding voor BitTorrent die peers toelaat om andere peers te vinden op het lokale netwerk (LAN). Hiervoor gebruikt het speciale multicast aankondigingen die niet buiten het lokale netwerk gerouteerd worden. Een peer kondigt aan welke torrent hij aan het downloaden is, zodat andere luisterende peers een verbinding kunnen openen met deze peer.

In de praktijk wordt deze uitbreiding weinig gebruikt in BitTorrent. Het is weinig waarschijnlijk dat twee gebruikers op een thuisnetwerk tegelijk dezelfde torrent zullen downloaden, en op een bedrijfsnetwerk wordt BitTorrent vaak niet toegelaten. In de context van draadloze sensornetwerken is het wel zeer relevant, en daarom is het opgenomen in aangepaste vorm in het NanoTorrent protocol.

B. Deluge

Deluge [11] is een protocol om bestanden te distribueren in een draadloos sensornetwerk.

Knopen kondigen periodiek aan welke versie van het bestand ze hebben, en luisteren naar meldingen van hun buren. Wanneer ze ontdekken dat een buur een nieuwe versie heeft, sturen ze een aanvraag voor de delen van dit bestand. Buren met delen van dit bestand kunnen dan hun data verspreiden naar geïnteresseerde buren.

Het gebruikt Trickle timers [13] om efficiënt lokale broadcasts te versturen naar buren. Iedere knoop luistert eerst een tijdje naar wat andere buren versturen alvorens zelf een bericht te sturen. Als ze merken dat verschillende buren reeds een identiek bericht hebben verstuurd, is het niet nodig dat de knoop dat bericht nog eens herhaald. De knoop kan zo besparen op zijn transmissie.

Deluge neemt dus gretig gebruik van de efficiëntie van lokale broadcasts in draadloze sensornetwerken. De ideeën kunnen ook gebruikt worden voor piece uitwisseling met lokale peers in NanoTorrent, maar vertalen moeilijk naar ver afgelegen peers gevonden via de tracker. Om het ontwerp van het prototype eenvoudig te houden, zijn de optimalisaties van Deluge voorlopig niet geïmplementeerd in NanoTorrent.

C. TinyTorrents

TinyTorrents [14] is een peer-to-peer protocol om torrents te verspreiden in een draadloos sensornetwerk. Het is sterk geïnspireerd door BitTorrent, en gebruikt ook torrent files, trackers en pieces.

De auteurs van TinyTorrents herkennen vele van de uitdagingen van draadloze sensornetwerken in de problemen die BitTorrent oplost. BitTorrent tracht de belasting op het netwerk evenwichtig te houden, verifieert de integriteit van pieces met checksums en maakt distributie snel en efficiënt met de zeldzaamste-eerste policy voor pieces. Mede daarom komen deze concepten ook terug in het ontwerp van NanoTorrent.

TinyTorrents biedt ook compatibiliteit met gewone BitTorrent door middle van een gateway die de brug spant tussen beide protocollen. Hoewel dit interessant is om sensormetingen beschikbaar te maken aan BitTorrent clients op het internet, is het niet echt nuttig om bestanden te sturen naar een draadloos sensornetwerk.

D. Uitdagingen en aanpakken voor het herprogrammeren van draadloze sensornetwerken

Wang et al. [15] schetsen een raamwerk om de verschillende functies nodig in een oplossing voor het herprogrammeren van sensornetwerken te onderzoeken. Hun framework omvat functies zoals versiebeheer, selectie van scopes, en aanvragen van code. Een protocol voor bestandsdistributie zoals NanoTorrent kan hierin de functie van code aanvragen en code distributie vervullen.

Hun paper onderzoekt ook bestaande protocollen zoals Deluge, en categoriseert ze volgens hun functies en toepasbaarheid in verschillende configuraties. Ze merken op dat weinig protocols toelaten om slechts een deel van het netwerk te herprogrammeren. Dit is één van de punten waar NanoTorrent probeert op te verbeteren, door slechts een deel van de nodes te laten deelnemen aan een torrent distributie.

VII. Conclusies

NanoTorrent verkent de mogelijkheid om twee manieren om peers te vinden te combineren. Peers kunnen lokaal buren vinden die dezelfde torrent aan het downloaden zijn, en met de tracker kunnen ze verbinden met ver afgelegen peers om pieces te verspreiden naar andere delen van het netwerk. Helaas brengt deze hybride aanpak een trade-off met zich mee: de snelheid van de distributie kan inderdaad verhoogd worden, maar dit gaat ten koste van meer berichten over het netwerk.

Doordat NanoTorrent bovenop IPv6 is geïmplementeerd, kan het communiceren over meerdere tussenliggende knopen en zelfs meerdere netwerken zonder extra inspanningen in het ontwerp van het protocol. Het protocol maakt gebruik van de bestaande routing protocollen aangeboden door alle knopen in het netwerk, in plaats van een nieuw aangepast routing protocol uit te vinden. Naarmate draadloze sensornetwerken groter worden, zal het nodig zijn dat meer protocollen op IPv6 kunnen werken om mee te kunnen schalen met de toepassingen.

Het protocol kan nog op veel plaatsen verbeterd worden, en meer evaluaties zijn nodig om ten volle de impact van de hybride aanpak om peers te vinden in kaart te brengen.

Referenties

[1] H. S. N. Lab. (2008) Volcano monitoring. http://fiji.eecs.harvard.edu/Volcano.

[2] O. Vermesan *et al.*, “Internet of Things beyond the Hype: Research, Innovation and Deployment,” in *Building the Hyperconnected Society - IoT Research and Innovation Value Chains, Ecosystems and Markets*. River Publishers, 2015, ch. 3, pp. 15–118.

[3] M. Chan, D. Esteve, C. Escriba, and E. Campo, “A review of smart homes - present state and future challenges,” *Computer Methods and Programs in Biomedicine*, vol. 91, no. 1, pp. 55–81, 2008.

[4] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke, “A Survey on Smart Grid Potential Applications and Communication Requirements,” *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 28–42, 2013.

[5] ——, “Smart Grid Technologies: Communication Technologies and Standards,” *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 529–539, 2011.

[6] M. Corporation. (2006) TMote Sky: Datasheet. http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf.

[7] L. Electromechanical. (2013, May) Zigduino r2 Manual. http://wiki.logos-electro.com/zigduino-r2-manual.

[8] B. Cohen. (2011, Oct.) The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html.

[9] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *29th Annual IEEE Conference on Local Computer Networks*, ser. Proceedings - Conference on Local Computer Networks. LOS ALAMITOS: IEEE Computer Soc, 2004, Conference Proceedings, pp. 455–462.

[10] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, “Cross-level sensor network simulation with COOJA,” in *31st Annual IEEE Conference on Local Computer Networks*, ser. Proceedings - Conference on Local Computer Networks. NEW YORK: IEEE, 2006, Conference Proceedings, pp. 641–648.

[11] J. W. Hui and D. Culler, “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale,” in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys ’04. New York, NY, USA: ACM, 2004, pp. 81–94.

[12] Wikipedia. Local Peer Discovery. https://en.wikipedia.org/wiki/Local_Peer_Discovery. Visited on 2015-07-25.

[13] P. Levis, N. Patel, D. Culler, and S. Shenker, *Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks*. USENIX Assoc., 2004, pp. 15–28.

[14] C. Mc Goldrick *et al.*, “TinyTorrents - Integrating Peer-to-Peer and Wireless Sensor Networks,” *Wons 2009: Sixth International Conference on Wireless on-Demand Network Systems and Services*, pp. 109–116, 2009.

[15] Q. Wang, Y. Y. Zhu, and L. Cheng, “Reprogramming wireless sensor networks: Challenges and approaches,” *IEEE Network*, vol. 20, no. 3, pp. 48–55, 2006.

Bibliography

[1] IEEE Standards Association. IEEE 802.15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). http://standards.ieee.org/getieee802/download/802.15.4-2011.pdf, 2011.

[2] M. Chan, D. Esteve, C. Escriba, and E. Campo. A review of smart homes - present state and future challenges. *Computer Methods and Programs in Biomedicine*, 91(1):55–81, 2008.

[3] Han Chih-Chieh, R. Kumar, R. Shea, and M. Srivastava. Sensor network software update management: a survey. *International Journal of Network Management*, 15(4):283–294, 2005.

[4] B. Cohen. The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html, October 2011.

[5] Moteiv Corporation. TMote Sky: Datasheet. http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf, 2006.

[6] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE Conference on Local Computer Networks*, Proceedings - Conference on Local Computer Networks, pages 455–462, LOS ALAMITOS, 2004. IEEE Computer Soc.

[7] Logos Electromechanical. Zigduino r2 Manual. http://wiki.logos-electro.com/zigduino-r2-manual, May 2013.

[8] Internet Engineering Task Force. RFC 4291: IP Version 6 Addressing Architecture. https://tools.ietf.org/html/rfc4291, February 2006.

[9] Internet Engineering Task Force. RFC 4862: IPv6 Stateless Address Autoconfiguration. https://tools.ietf.org/html/rfc4862, September 2007.

[10] Internet Engineering Task Force. RFC 6282: Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. https://tools.ietf.org/html/rfc6282, September 2011.

[11] Internet Engineering Task Force. RFC 6550: RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. https://tools.ietf.org/html/rfc6550, March 2012.

[12] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems-the International Journal of Grid Computing and Escience*, 29(7):1645–1660, 2013.

[13] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke. Smart Grid Technologies: Communication Technologies and Standards. *IEEE Transactions on Industrial Informatics*, 7(4):529–539, 2011.

[14] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke. A Survey on Smart Grid Potential Applications and Communication Requirements. *IEEE Transactions on Industrial Informatics*, 9(1):28–42, 2013.

[15] D. Hughes et al. LooCI: the Loosely-coupled Component Infrastructure. *2012 11th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 236–243, 2012.

[16] Jonathan W. Hui and David Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 81–94, New York, NY, USA, 2004. ACM.

[17] Harvard Sensor Networks Lab. Volcano monitoring. http://fiji.eecs.harvard.edu/Volcano, 2008.

[18] P. Levis, N. Patel, D. Culler, and S. Shenker. *Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks*, pages 15–28. USENIX Assoc., 2004.

[19] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[20] C. Mc Goldrick et al. TinyTorrents - Integrating Peer-to-Peer and Wireless Sensor Networks. *Wons 2009: Sixth International Conference on Wireless on-Demand Network Systems and Services*, pages 109–116, 2009.

[21] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *31st Annual IEEE Conference on Local Computer Networks*, Proceedings - Conference on Local Computer Networks, pages 641–648, NEW YORK, 2006. IEEE.

[22] Sandvine. Global Internet Phenomena Report: 2H 2014. https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/2h-2014-global-internet-phenomena-report.pdf, 2014.

[23] O. van der Spek. UDP Tracker Protocol for BitTorrent. http://www.bittorrent.org/beps/bep_0015.html, March 2015.

[24] O. Vermesan et al. Internet of Things beyond the Hype: Research, Innovation and Deployment. In *Building the Hyperconnected Society - IoT Research and Innovation Value Chains, Ecosystems and Markets*, chapter 3, pages 15–118. River Publishers, 2015.

[25] Q. Wang, Y. Y. Zhu, and L. Cheng. Reprogramming wireless sensor networks: Challenges and approaches. *IEEE Network*, 20(3):48–55, 2006.

[26] Wikipedia. Local Peer Discovery. https://en.wikipedia.org/wiki/Local_Peer_Discovery. visited on 2015-07-25.

Fiche masterproef

Student: Mattias Buelens

Titel: NanoTorrent: Locality-aware peer-to-peer file distribution in wireless sensor networks

Nederlandse titel: NanoTorrent: Plaatsbewuste peer-to-peer bestandsdistributie in draadloze sensornetwerken

UDC: 621.3

Korte inhoud:

Wireless sensor networks consist of many small computer devices equipped with various sensors and actuators which communicate with each other using their wireless antenna. In order to keep them operating for long periods of time, they must be able to adapt and evolve even after being deployed. This requires them to retrieve large files such as new programs or configurations over the network fast and efficiently.

NanoTorrent is a peer-to-peer protocol that allows fast distribution of files in wireless sensor networks. It introduces the use of a hybrid peer discovery mechanism with both a tracker and local neighbour discovery, allowing it to work even in heterogeneous deployments with nodes running different programs. The peer-to-peer approach helps balance the load of the file distribution on the seed and the network, and takes advantage of link-local multicast messages to distribute parts of the file to many neighbours simultaneously.

The evaluation of the protocol shows that NanoTorrent can provide fast file distribution in many different network configurations. The hybrid peer discovery approach comes with a trade-off, where increased distribution speed also comes with additional transmissions taking multiple hops across the network to distribute the file to distant nodes.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Gedistribueerde systemen

Promotor: Prof. dr. Danny Hughes

Assessoren: Dr. Sam Michiels
Dr. ir. Jan Ramon

Begeleiders: Dr. Nelson Matthys
Wilfried Daniels