FileCourier: Ad-Hoc Filesharing

Thijs Meijerink, Bär Halberkamp, Steven Raaijmakers
University of Amsterdam

February 5, 2017

I. Introduction

The ubiquity of smartphones equipped with Bluetooth connectivity gives rise to interesting networking possibilities. Besides the well known one-on-one communication, the technology allows for more elaborate wireless ad-hoc networks to be formed. To explore the possibilities of such networks, we have developed an app that facilitates filesharing between mobile phones using the Bluetooth protocol: FileCourier.

The scenario FileCourier aims to support, is a group of users centered around a certain location or event (e.g. a university campus, a conference) who want to share files with each other. Because the range of a typical Bluetooth device is about 10 meters [8], the users' mobile phones will typically not be able to form a connected network. In other words, there may not be an end-to-end path between two nodes that want to share data. *FileCourier* finds the best solution for sending a file between two disconnected parts (a *component*), inside a Bluetooth network, using a device as a courier.

II. Component

Each component can be viewed on its own, because such a component actually is a Bluetooth-Network itself, containing multiple Bluetooth-devices (*nodes*). A node in the network can complete two different tasks: being a master or slave [7]. A node typically becomes a master when it successfully initiates a connection to a *remote device*. This remote device will become a slave-node once it accepts the incoming connection.

This master/slave relationship is handled via two different threads:

- *Connect-thread*: initiates a connection
- *Listen-thread*: accepts an incoming connection

The master tries to connect its connect-thread to the listen-thread of the slave. The Bluetooth protocol allows a master to connect with up to 7 different slaves, and states that it's possible for a slave to be a master of another device, or a slave to multiple masters, thus creating a directed network.

The Bluetooth network implemented in FileCourier however does not support the possibility for a node to fulfill both tasks (master/slave) simultaneously. Therefore each component can be seen as a one *piconet* [1].

Whether a device joins as a slave or master depends on its user. Before joining the network, the user can choose between joining as either a master or a slave node. Joining as master-node therefore means creating a new network, instead of joining an existing one, which will happen when joined as slave.

i. Autoconnect via UUID

Autoconnecting between devices in FileCourier is handled via UUID. The *UUID* is a unique identifier used in the Bluetooth-threads [3]. When an UUID is specified for a listen thread, it will only accept connections with connect-thread where the UUID matches.

A master device knows which devices it can connect to, by discovering all the Bluetooth-devices within its range. When a master-device is finished discovering, it will automatically try

to connect to all those devices, while only devices with a matching UUID will accept the incoming connection.

ii. Forwarding

Since each slave is only connected to its master, it is not possible to communicate directly between slaves in the same piconet. Whenever a slave wants to communicate with another slave, the master has to forward the messages between these two slaves, creating a virtual connection between two slaves.

The forwarding is handled by the master comparing the receiver-field of the incoming message. Whenever the receiver-address does not match the master's own address, the master determines how it should forward the message. This can happen in two ways:

- The receiver-address of the message matches one of the addresses of the master's slaves, and thus the message is forwarded to this device.
- When receiver-address is not found among the other slave-addresses, the master will determine which of all the devices in the piconet is most likely to connect to the supposed receiver via the developed algorithm discussed later.

iii. BluetoothMessage

The messages exchanged between two nodes are called *BluetoothMessages*. A BluetoothMessage contains 4 fields of information: sender, receiver, type and payload (containing the actual data to be sent).

Different type of BluetoothMessages can be sent, using the forwarding-algorithm. Various examples of these types *type* include *NETWORK_INFO*, *TOPOLOGY* and *FILE* (all discussed later).

NETWORK_INFO

The *NETWORK_INFO*-type message is used to distribute information about the current devices in the piconet. This information contains the addresses and names of all the devices in the current piconet. Thanks to this information, a slave knows which slaves it can virtually connect to. The distribution of this info is handled at the master node:

- Whenever the master successfully initiates a connection to a slave, it will update its own *connectedAddresses*, containing all the addresses and names of its current slaves.
- After this update the master will flood its updated connectedAddresses to all its slaves (including the newly joined slave).
- When a slave receives a NETWORK_INFO message, it will updates its own connectedAddresses to the received connectedAddresses, now containing the names and addresses of the other slaves in the piconet.
- Whenever a slave disconnects from its master, the master will again update the connectedAddresses by removing the disconnected slave. After the update, this information is again flooded to its slaves.
- When a master leaves the network, all the connectedAddresses of the former network participants will be reset.

III. Research

Routing between two disconnected components inside a network is characterized as a *Delay Tolerant Network* (DTN) [5]. Routing in a DTN presents additional challenges, which this project aims to address. Here follows an overview of DTN routing strategies encountered in the literature.

i. Routing in a DTN

DTN routing strategies can be roughly divided into two groups: *flooding* and *forwarding* strategies [2]. Flooding strategies propagate multiple copies of a file over the network, hoping that one will eventually reach the destination. Forwarding strategies use information about the network topology to determine a single, optimal path for a file to take.

ii. Flooding strategies

ii.1 Two-hop relay

In the two-hop relay algorithm, the node that wants to send a file copies it to the first n nodes it encounters [2]. For the file to be delivered, one of these nodes must eventually encounter the destination node.

ii.2 Tree-based flooding

Tree-based flooding is a generalized version of two-hop relay, that allows nodes that received a copy of a file to also distribute copies themselves [2]. The algorithm is tuned by limiting the number of copies each node makes (breadth of the tree), and the number of times a file may be copied (depth of the tree). By tuning these parameters, a compromise can be made between the consumption of network resources, and the probability that the file reaches its destination, and how quickly it does this.

ii.3 Epidemic routing

Epidemic routing is the most extreme form of flooding, where a file is distributed across all nodes at the first possibility [2]. When two nodes meet, each gives the other the files that it does not have. This guarantees that a file will reach the destination as quickly as possible (provided enough network resources are available), but wastes a lot of bandwidth on distributing files to nodes that have no interest in them.

iii. Forwarding strategies

iii.1 PRoPHET

The *Probabilistic Routing Protocol using History of Encounters and Transitivity* attempts to exploit the predictability of human movement patterns [4]. Each node calculates a *delivery predictability* metric for each known destination node, expressing the probability of succesfully delivering a file to that destination. When two nodes meet, they exchange their metrics and update their own metrics according to this new information.
The metric is increased whenever nodes meet each other, and decreases with time. The metric is also transitive, meaning that if node B has a high delivery predictability for node C, and node A has a high delivery predictability for node B, then node A will also have a high delivery predictability for node C. The amount of influence this transitive property has on the delivery metric is determined by a multiplicative factor, thus allowing network designers to encode a preference for single- or multi-hop routing.
The authors of the PRoPHET protocol don't specify how this metric must be used for routing decisions, and instead propose that the forwarding behaviour is tweaked for individual applications.

iii.2 Gradient routing

In the gradient routing (GRAd) protocol, a node carrying a message broadcasts its own cost for delivering the message to its neighbours. The message is forwarded to all nodes that have a lower cost for delivery to this destination [6]. Nodes also update their own cost function from these broadcasts: a message carries with it the cost (e.g. number of hops) from its point of origin, and each node that receives the broadcast updates its own cost function with respect to this point of origin accordingly.

IV. FileCourier's routing algorithm

Previous projects in the net-centric computing course have delivered files or messages by flooding them to all neighbours. As an improvement, we want to deliver files with minimal load on the network. For this reason, we use a single-copy routing protocol. Besides the lower network load, this has the advantage that when a file has reached the destination, no further work needs to be done to stop the spreading of file through the network.
The routing protocol consists of the following components: a connectivity score, and a topol-

ogy data structure. The topology data structure contains the mutual connectivity scores of all nodes, and is used to propagate this information through the network.

i. Connectivity score

The connectivity score is a measure of how often two nodes are in the same piconet. Because we want this score to be up-to-date, but also to reflect events that happen once a week (e.g. a certain weekly lecture), the connectivity score is calculated based on the connection history of the past two weeks.
When two nodes are in the same piconet, they are considered connected, because a message can be sent directly between them. When a piconet changes (a node enters or leaves) this information is sent by the master node to all slaves. When the master node becomes unavailable, slaves themselves will notice this. Using these updates, each node keeps track of the connection history with the other nodes. The most recent two weeks of this history are converted into the percentage of time these nodes were connected, stored as a floating point number between 0 and 1. This is the connectivity score.

ii. Sending a message

If the destination node of a message is not found within the current piconet, the sender uses its topology data structure to determine the shortest path to the destination node. The score of a multi-hop path is calculated by multiplying the connectivity scores of all links. As soon as the sender is within the same piconet as the next node in the shortest path, it forwards a message (containing the file and the calculated shortest path) to this node. This node then forwards the message to the next node in the shortest path as soon as it becomes available, until the message has reached the destination.

V. Topology

The topology can be seen as the beating heart of the routing algorithm. By combining and parsing the topologies received by neighbors, each node builds up "their" topology, which it uses for calculating the shortest path to a destination when sending files. To build up a proper topology, there are a few requirements:

- Each node must only be in the topology once. This is essential for avoiding cycles, as well as simplifying the `get` actions.
- Each node must also have a single path leading to it. Storing alternate paths can lead to infinite loops when calculating the shortest paths, and adds completely redundant information to the topology, as paths leading to a certain node for which a higher scored path already exists are ignored by the algorithm.
- It must be easy to remove old subtopologies received by neighbors and replace them with new ones. Since each node only personally keeps track of the score with their neighbors, nodes rely entirely on received data to complete their topology. Since the layout of an entire "branch" of the topology can change with a single update, each update needs to remove and rebuild the entire topology received by the neighbor.
- The entire data structure must be easily converted to JSON to allow transmission to other nodes.

Keeping these requirements in mind, we implemented the topology as a tree data structure. The root node consists of the device itself, with a branch for each neighbor. Because of how a tree works, these branches are in turn the root of their own tree: the topology received from the neighbors in the network. Prevention of cycles and duplicate nodes is accomplished using the following algorithm:

```
# Python pseudo-code implementation of the node addition algorithm used
# in the java code. In this case 'self' refers to the entire tree,
# whereas 'tree' refers to the subtree being added.
def add(self, tree, score_so_far):
        node = tree.root
        score = score_so_far * node.score
        # total score to a node is the product of the scores in
        # between nodes on the path

        if max[node.name] and max[node.name] >= score:
                return
                # This means the node is already in the tree with a higher score,
                # and the algorithm should stop.
        else if max[node.name] and score > max[node.name]:
                # This means the node is already in the tree, but with a lower score
                # Which means the old node should be deleted,
                # and this node should be put in its place
                delete_node(self.find_node(node.name))
                max[node.name] = score
        else:
                # This means the node isn't already found.
                max[node.name] = score

        add_node(node)

        for child in node.children:
                self.add(child, score)
```

The topology can be exported to JSON. This is both useful when saving the topology, as well as encoding the topology into a BluetoothMessage.

This algorithm relies on a `max` variable, which algorithm will be discussed next.

i. Caching

To allow usage of the topology without having to transverse the entire tree every time and to keep track of which files are currently in the process of being sent, we utilize caching in various locations throughout the networking code.

As mentioned previously, the topology class utilizes a `Map` with maximum scores for each node, as well as the respective shortest path. If a value is requested using a call of the `get()` function, this cache is first checked. If the value isn't found, the tree is transversed, and the found value saved in the cache. Newly added nodes are immediately added to the cache as well.

Another place where caching is utilized is when sending files. Due to the instability in the topology, the node the file is sent to is determined before starting the transmission, and is cached afterwards. This way the topology is no longer needed, as only the next hop is necessary information, since each "relay node" calculates its own path to the final destination (also due to the aforementioned instability in the topology). Caching is also utilized to save the list of file chunks that still need to be sent. In the current architecture, only a single node sends files to a single receiving node. Caching the progress of the transmission limits the amount of information that needs to be sent over the network, which can therefore utilize a very simple protocol.

VI. Files

Files in the app are handled by the `FileManager` and are divided in two categories, `Files` and ReceivedFiles. While these types both refer to a file, and both use a `File` object under the hood, they have certain unique properties. Both are saved in separate `.json` files, each with their own array of objects, containing different fields depending on the type of file. As well as the list of files, `FileManager` contains a `Map` of destinations per file, signifying which file needs to be sent where, enabling easier addition and deletion of destinations. This `Map` is not saved separately, however. By saving the destination with the files, the pruning code that checks whether the files you're saving/loading still exist will also prune the destination list, limiting space taken up by missing files being mentioned still.

"Normal" `Files` refer to files the user has uploaded into the app by themselves. They're loaded in with a path, and have a filename and destination attached to them. However, files do not use their original name after being uploaded into the app. Since filenames can be chosen arbitrarily, file collisions are very likely to happen. To mitigate, a new filename is generated upon upload, and used for all operations within the app. Because the entropy in a (long) random string is significant enough, this allows sending files without checking for duplicates, which simplifies communication. The original filename is saved alongside the new one, still. This is done to make files easier to recognize for users of the app. The extension of the file is also extracted from the original filename, and used to detect the MIME type of the file, allowing the receiver to open the file. (Android does not correctly detect the MIME type of a file without an extension.)

A `ReceivedFile` contains most of the attributes found in the "normal" file, minus the original filename, as this isn't sent by the network. It does however contain support for dividing the file in chunks. Upon receiving a new file, the file is divided in chunks of 64kb, allowing the sender to send the file in parts, which in turn allows easier pausing and resuming of transmissions, and allows a receiver to receive a file from multiple sources at once, although this is not supported in the app at this time.

When initializing a new `ReceivedFile`, the file is first filled with empty bytes, equal in size to the final file. This doubles as memory allocation, and allows chunks to be received out of order. To keep track of which chunks are and aren't received, the file keeps track using a 64-bit bitmap. This also means the maximum filesize for files in the app is `64kb * 64 = 4096kb`.

i. File protocol

For communication during file sharing, a simple protocol is made, allowing the messages to consist of entirely strings, which makes parsing simpler. There is no input validation as of now. This is both done due to time constraints, and the fact that Bluetooth requires explicit permission before connecting nodes, which makes it much harder for malicious parties to cause damage.

To send a file, a `FILE` message is first sent. This message is of the form '`fname + "|" + destination + "|" + size`' with fname being the filename, destination being the final destination of the file, and size being the size of the file in bytes. The receiving party will then create the file if needed, and respond with an `ACK` message, containing the filename and bitmap of chunks the receiver already has, in the form of a `long`. The sender will then proceed to cache the info in this `ACK`, and proceed with sending each needed chunk of the file in `FILE_CHUNK`, one by one, until all needed chunks are received and parsed. Upon successfully receiving and parsing a chunk, the receiver will respond with a `CHUNK_ACK` message containing the number of the chunk received. The sender will not send another chunk until this `CHUNK_ACK` is received, and will give up on sending the file if no `CHUNK_ACK`s are received.

VII. Discussion

Future research should focus on the empirical performance of our application in practical scenarios, and possibly tweaking the algorithm to improve the precision of determining the optimal courier device.

It would also be interesting to create a hybrid of the presented algorithm and multi-copy routing algorithms, by spreading the file to multiple couriers, instead of just one. The balance between a lower network load and an increased chance of a successful or quicker delivery can be an area of further research.

During implementation and testing of our application, some Bluetooth related problems were encountered. Bluetooth might not be the most ideal network protocol for the implementation of an ad-hoc filesharing application. Choosing e.g. ad-hoc WiFi over Bluetooth could possibly have made the end results more appealing, if it would provide a smoother networking experience.

However, Bluetooth is a widespread and promising technology for ad-hoc networking, and with this project we have shown that it is possible to overcome its limitations and use it to build a usable file sharing application.

VIII. Conclusion

In this project, we implemented a Bluetooth file sharing application for disconnected ad-hoc networks. In comparison with existing research in ad-hoc networking, our application reduces load on the network by calculating a single optimal path for the data, and achieves fast transmission by exploiting the predictability of human movement.

References

[1] Jennifer Bray. "Masters and Slaves: Roles in a Bluetooth Piconet". In: (2001).

[2] Evan PC Jones et al. "Practical routing in delay-tolerant networks". In: *IEEE Transactions on Mobile Computing* 6.8 (2007).

[3] Peter Leow. *Android Connectivity*. `https://www.codeproject.com/Articles/814814/Android-Connectivity`. 2014.

[4] Anders Lindgren, Avri Doria, and Olov Schelén. "Probabilistic routing in intermittently connected networks". In: *ACM SIGMOBILE mobile computing and communications review* 7.3 (2003), pp. 19–20.

[5] Biren Patel and Dr Vijay Chavda. "Comparative study of DTN routing protocols". In: *International Journal of Advanced Research in Computer and Communication Engineering* 4.5 (2015).

[6] Robert Poor. *Gradient routing in ad hoc networks*. 2000.

[7] Paul; O'Connor Gerald; Connaughton Paul Tsang Will; Carey. "Bluetooth Terminology". In: (2001).

[8] Roger M Whitaker, Leigh Hodge, and Imrich Chlamtac. "Bluetooth scatternet formation: a survey". In: *Ad hoc networks* 3.4 (2005), pp. 403–450.