Assignment Documentation

Simple Device Discovery Protocol

Arto Suvitie
Mats Jalas

Current implementation status

The application is currently fully functional, however some improvements were not implemented due to time constraints.

Changes to the protocol

No drastic changes have been made to the overall protocol design.
At one point we decided to move the functionality of sending description request messages from input manager class to description manager class. The original reasoning was that description requests are triggered by user input but later we realized that it makes more sense to have all description message related functions under the Description manager class. Also, at one point we designed a separate class for listing outgoing description requests so that we could track which messages sent by the input manager were answered back in the description manager. This class was removed when we remembered that we were implementing description messages over TCP and thus the responses to description messages are received in the same function that sent the request.
At one point we realized that our design used only a single socket for TCP connections, which could cause problems if multiple description requests arrived simultaneously. We fixed this minor oversight by making the description listener run in separate threads with separate TCP sockets.
We had originally planned on using a separate port for tcp and udp sockets. However, we realized that this was not actually necessary or even useful so we decided to use just one port per node.
We also thought that since the hub has a specific port assigned to it, that we could determine any packets sent through the hub simply by looking at the port of the incoming packet's address. Of course, the port displayed there is not the port that was used to send the message, which is why we will have to include information about whether or not the packet was sent to the hub in the packet payload.

Current challenges

We managed to overcome all major challenges during this project. The last challenge we were working on was to implement as many improvements and optimizations as we could and to get measurement data of the performance of our protocol. The improvements we could not implement due to time constraints are listed at the end of the decision diary and the measurements are shown in a later chapter.

Decision diary

Implementation history:

12.2.2015
Implemented message and service classes with container classes for both. No changes were made to design.

23.2.2015
Implemented manager classes for descriptions and discovery message handling. Still no changes in protocol nor prototype design.

2.3.2015
Added node main loop, which also is the application main loop. Main loop creates all the managers and starts the select loop for handling incoming input from standard input and sockets.

3.3.2015
Many name and description improvements were made to make the code easier to read and maintain. Changed description and discovery manager classes to handler classes, which only creates and parses messages. Added broadcast discovery loop class to handle discovery message broadcasting, which uses discovery handler for message creation and modification. Added discovery listener for listening to incoming discovery messages and handle them accordingly.

We noticed that our design was missing multiple TCP sockets for sending and receiving descriptions messages, and by adding a TCP socket for each case might be tedious to implement in the select loop.

Decided to keep the select loop and implement the message handling parts in the background. First version of the implementation utilizes threads for the background work, and queues for messaging between threads.

5.3.2015
We realized that our Discovery Broadcast Loop has no way of knowing whether or not the Hub exists. Discovery Listener will have to determine whether or not the Hub is available and then communicate that info to the Broadcast Loop.

10.3.2015
Instead of having the Discovery Listener determining hub availability, it now only tells Discovery Broadcast Loop whenever there has been a new message received from the hub. The broadcast loop itself will then update a timestamp, which is used to determine whether the discovery message should be sent to the hub only or also to all available ports. The discovery message is sent to the hub even when the hub has timed out. This allows the program to recover from a scenario where the hub is temporarily unavailable. We also have plans for implementing separate smaller messages to be used for the p2p messaging in case our regular discovery packets were to cause congestion or other problems due to their size.

31.3.2015
It turned out that we had been using an unbound socket for sending udp messages. After making the discovery broadcast loop use our listening udp socket also for sending messages, we were able to get the udp hub working.

11.4.2015
We implemented active discovery which allows the network to discover nodes faster. This means that a node will immediately respond when it receives a discovery message from a previously unknown node.

26.4.2015
We optimized the port scanning function. Now, when the hub is not available, the node will only do a port scan every tenth time the broadcast loop runs. Other times, the node will only message its known peers. This improvement considerably reduces network flooding in the absence of a hub, while still maintaining connections between nodes.

28.5.2015
The program works now and we have managed to get some measurement data as well. However, there were several planned improvements which we did not have time to implement.
The following improvements were dropped due to time constraints:

- Creating unit tests for all classes and methods
    - We did create tests for some of the more important parts of the program but didn't have time to do it for everything. This improvement would have made further development and debugging easier.
- Implementing message types
    - We planned having message types included in the message payloads. The program works just fine without this feature as there's only three message types which can be identified by the used transport protocol and contents. However, this feature would have been useful for further development and optimization.
- Improving message contents

    - We had planned on leaving service information out of description request messages. This feature was done and almost implemented, but it had to be dropped as it would not work without the message types. As the program is currently, it does not distinguish between description and discovery messages. Therefore, leaving service list out of the message would have caused the program to misinterpret that the sender had lost all their services.
- Implementing security measures
    - We had discussed implementing some kind of methods for verifying message integrity. This feature was dropped because it was not essential for the performance of the program.
- Adding/removing services to local node at runtime
    - The program fully supports peer nodes adding, removing or updating their services. However, it does not currently have any mechanism for manipulating the node's own services at runtime. This means that any changes to service list require the node to be shut down and restarted.
- Implementing a method for toggling output between stdout and logfile
    - This feature would have allowed for cleaner testing of multiple nodes simultaneously. However, our testing framework mostly obsoleted this feature.
- Support description requests for single service
    - Currently our program fully supports a description request, which is answered with descriptions of all services. We thought it might be good to also support requests that would ask for the description of only one service.This improvement would have slightly lowered network load in the case where nodes have huge amounts of services but are only interested in the descriptions of one or two services.

Prototype architecture

The architecture is visually illustrated in appendixes A and B. The prototype consist of one main loop, which listens to incoming messages from other nodes and to commands from a user or application utilizing the protocol. Before the listening loop is started, a background job is created for broadcasting discovery messages to other nodes in the network. This job continues sending discovery messages in a specified time interval as long as the node is alive. It also cleans up the list of the other nodes when their last sent discovery message has not arrived before the clean time intervall.

In the main loop two sockets are created, one for incoming TCP connections and one for UDP messages. These two sockets and the standard input file descriptor is added to a input list, which is given to the select function. The select function enable incoming connection and messages to run in the same process, so the architecture is kept simpler.

When a TCP connection is requested a new socket gets created to handle that connection and given to a DescriptionHandler object that can run its procedures in a own thread. This way the main loop can focus on listening to connections and messages and delegate handling procedures to other entities. Same logic is used for incoming UDP messages that contains incoming discovery messages from other nodes. When a user or a third-party application sends commands to the application, then also handling of these commands are delegated to a separate thread.

We have been striving to keep the prototype as modular as possible, so that it is easier to make changes and maintain the prototype code. We have also strived to keep the classes as small as possible, and to let one class focus on only one task. Also we have tried to create meaningful class and method names so that the code can be easily understood and that it should be easier to fix bugs. We have created some unit tests to test some of the classes, but more tests need to be created so that code changes can be made without unknowingly breaking other code.

The program will support p2p type of communication whenever the hub is not available. The Discovery Broadcast Loop that handles sending discovery messages to the hub will use a timestamp to determine if it should send the discovery message to all available ports. This timestamp is updated whenever the Discovery Listener receives a message from the hub.

Protocol Measurements

We performed several measurements to find out how efficient our program is. We measured the following metrics:

- Network discovery time
    - Tells us how fast a new node is able to discover all other nodes in the network.
- Node discovery time
    - Tells us how quickly a node connecting to a network is detected by existing nodes.
- Node disappearance detection time
    - Tells how quickly other nodes detect the disappearance of a node
- Protocol overhead:
    - Tells us the ratio of total network bytes sent in comparison to useful information received during discovery and description processes.

We obtained the following results:

Network discovery and disappearance times

----4 node network + test node----
New node discovered nodes in network:
Node "test_node" discovered all in 0:00:06.026396

Node disappearing discovery times (9 second timeout limit):
'test_node': min: 0:00:07.423269, max: 0:00:07.437479

----10 node network + test node----
New node discovered nodes in network:
Node "test_node" discovered all in 0:00:06.047748

Node disappearing discovery times (9 second timeout limit):
'test_node': min: 0:00:11.367294, max: 0:00:11.403695

----20 node network + test node----
New node discovered nodes in network:
1) Node "test_node" discovered all in 0:00:06.093558
2) Node "test_node" discovered all in 0:00:06.095488

Node disappearing discovery times (9 second timeout limit):
'test_node': min: 0:00:10.939268, max: 0:00:10.987093
Node disappearing discovery times (20 second timeout limit):
'test_node': min: 0:00:17.111592, max: 0:00:17.163839

----50 node network + test node----
New node discovered nodes in network:
1) Node "test_node" discovered all in 0:00:06.164957
2) Node "test_node" discovered all in 0:00:06.122424
3) Node "test_node" discovered all in 0:00:06.469219

Node disappearing discovery times (9 second timeout limit):
'test_node': min: 0:00:09.170152, max: 0:00:09.371838
Node disappearing discovery times (20 second timeout limit):
'test_node': min: 0:00:20.532211, max: 0:00:20.722507
Node disappearing discovery times (20 second timeout limit):
'test_node': min: 0:00:19.671285, max: 0:00:19.815136

Protocol overhead

There are many ways to interpret protocol overhead and how it should be calculated. We decided it would be useful to look at the packets we send and compare the useful bytes to all the sent bytes.
An example discovery message packet consisted of 294 bytes

Bytes	Content
42	(Packet headers)
252	{"address": {"port": **9299**, "ip": "**127.0.0.1**"}, "name": "**Node9299**", "timestamp": **1432838276.827814**, "services": [{"type": "**media**", "name": "**movie library**"}, {"type": "**sensor**", "name": "**temperature service**"}, {"type": "**media**", "name": "**music streamer**"}]}

Since the message payload is written in json, there is quite a bit of overhead when comparing to the optimal case. The payload in this example case has 152 bytes of pure overhead and only 100 useful bytes. The highlighted parts of the payload are the actual useful information. The parts highlighted in green are text strings that can be considered as useful bytes. The parts highlighted in yellow are numeric values which are sent as text, and therefore they have

room for optimization. The parts highlighted in orange are information that is not used by the current version of the program and therefore are currently redundant information. The optimal message would be sent in binary instead of json.
The IP could be optimized to use 4 bytes instead of 9, however, it could also be removed entirely as the program currently only supports local networks. The port could be optimized to use 2 bytes instead of 4 and the timestamp could be optimized to use 8 bytes instead of 17. Therefore, there's only 80 useful bytes in this example message (70 bytes in text strings, 2 bytes in port number, 8 bytes in timestamp, 0 bytes in ip). This gives us a protocol efficiency of 80/294 = 27%, which means the overhead is 73%.

In our example, the description response message is 584 bytes long.

Bytes	Content
66	(Packet headers)
518	{"timestamp": **1432893989.753976**, "address": {"port": **9931**, "ip": "**127.0.0.1**"}, "name": "**Node9931**", "type": "**description_response**", "services": [{"**description**": "**This is a temperature sensor service that collects temperature data.**", "name": "**temperature service**", "type": "**sensor**"}, {"description": "**This is a music streaming service.**", "name": "**music streamer**", "type": "**media**"}, {"description": "**This is a movie library service that provides movie browsing and streaming.**", "name": "**movie library**", "type": "**media**"}]}

Here, there's 221 purely redundant bytes. Again the timestamp, port and IP could be optimized to save 20 bytes, also the message type could be reduced to 1 byte instead of 20. Therefore, there's 221+20+19 = 260 redundant bytes, which leaves 258 useful bytes. This means that the packet efficiency is 258/584 = 44%, which means the overhead is 56%.

The description request message in our example consists of 211 bytes and is as follows:

Bytes	Content
66	(Packet headers)
145	{"address": {"ip": "**127.0.0.1**", "port": **9931**}, "services": [], "name": "**Node9931**", "timestamp": **1432893989.752229**, "type": "**description_request**"}

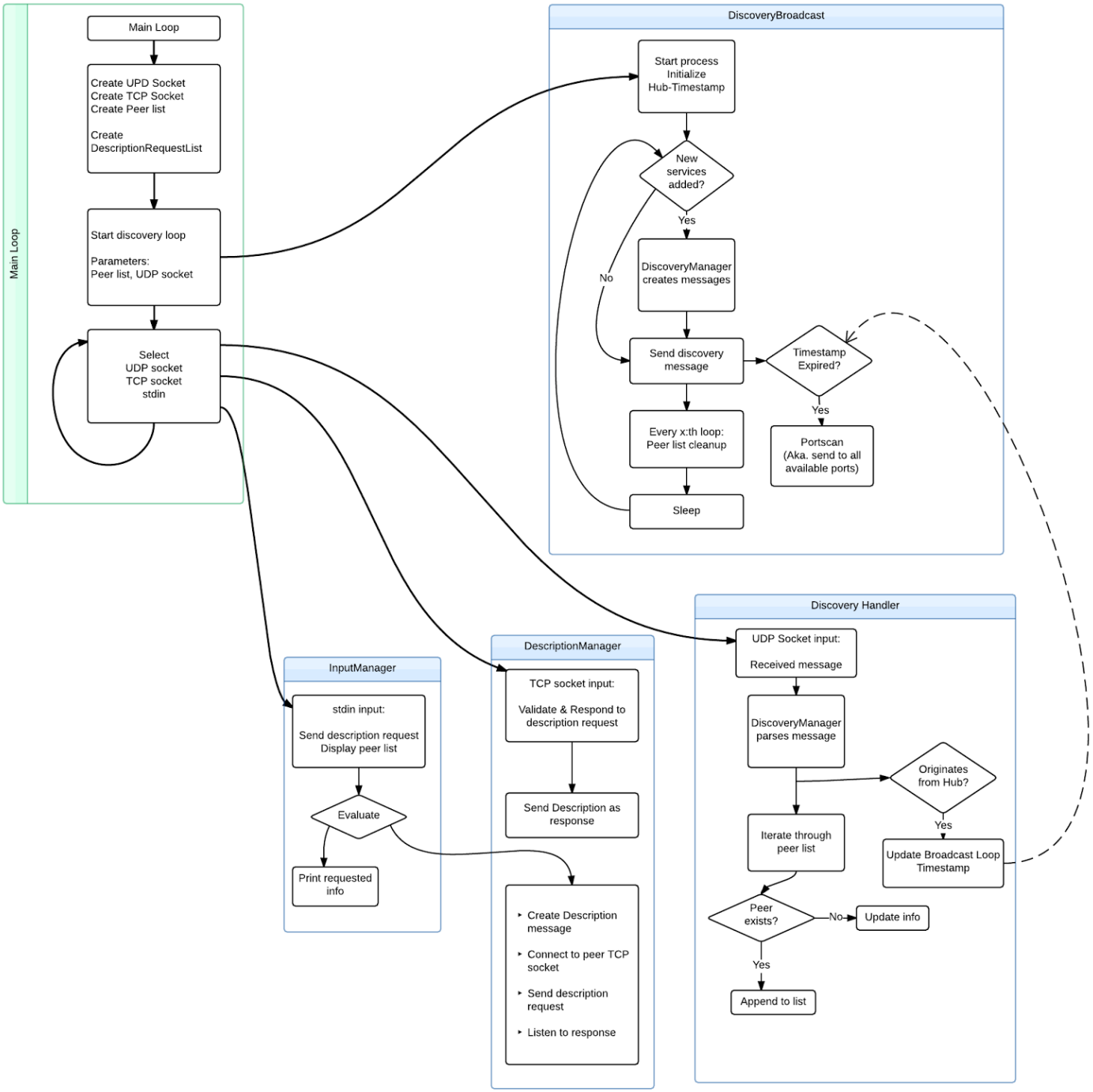
In this message, there's 4+2+8+1 = 15 useful bytes. In this case the node name is actually not used for anything. Packet efficiency is 15/211 = 7%, which means the overhead is 93%.

The complete description request and response process is as follows:

Bytes	Useful bytes	Packet	Sender
74	0	SYN	Requesting node
74	0	SYN, ACK	Answering node
66	0	ACK	Requesting node
211	15	PSH, ACK	Requesting node
66	0	ACK	Answering node
584	258	PSH, ACK	Answering node
66	0	ACK	Requesting node
66	0	FIN, ACK	Answering node
66	0	ACK	Requesting node
66	0	FIN, ACK	Requesting node
66	0	ACK	Answering node
1405	273		

So the efficiency of this entire transaction is 273/1405 = 19%, which means the overhead is 81%.

Appendix A: Flow chart



Appendix B: UML Chart