Routing in Ad Hoc Networks to Discourage Selfish Nodes

COMP 4203 - Final Report

James Tyler (100976157), Bradley Koon (100972222), Imdad Ali (101099711)

Date: April 5, 2018

# 

[**Introduction**](#_6p4l7f1gs1x1) **3**

[**Background**](#_y2ki0ob64tm3) **3**

[**Problem and Solution**](#_g1k98xcksrda) **5**

[**Analysis**](#_m83cxcpbf16g) **9**

[**Conclusion**](#_gkxzq0sji8nn) **12**

[**References**](#_2gwjpzz573yf) **13**

[**Packages Used**](#_qsk29c6ungwp) **13**

[**Setup Instructions**](#_a3wxaqy61l4i) **13**

# 

# 

# 

# 

# 

# Introduction

An ad hoc network is a collection of mobile nodes that wirelessly communicate with each other without the use of any fixed communication infrastructure. They are referred to as “MANETS”. Mobile nodes can act as routers to forward messages from a source to a destination. Ad hoc networks have been becoming increasingly popular over the years and are most commonly being applied in military communication, vehicular communications (VANETS), smartphone communication, bluetooth, etc. Ad hoc networks are among the most favorable types of networks since they provide better mobility, quick setup, flexibility in location, and are not dependent on any fixed infrastructure benefiting the cost of installation. Since mobile nodes operate wirelessly, they have limited battery lives and spend a lot of energy forwarding messages during route discovery, route maintenance and routing. Mobile nodes with significantly low battery levels will resort to becoming “selfish” by refusing to forward packets in order to conserve their energy. The presence of selfish nodes can greatly disrupt network communication. This can affect the quality of calls and can be especially fatal in life threatening situations such as war zones where a communication delay of just a few minutes can mean a person’s life. For a well functioning ad hoc network, it is important for mobile nodes to be aware of selfish behaviors and find ways to discourage it. The Optimal route must also be selected based on the path with highest successful delivery probability as opposed to the shortest path.

For our project, we modified the SA-DSR routing protocol proposed in [2] using selfish node detection techniques described in [1] in order address the shortcomings of SA-DSR and help to discourage selfish behaviour of nodes in MANETs.

# Background

To explore the idea of selfish node detection in a little more depth, we read two research papers. The first paper [1] discusses the idea of selfish detection in ad hoc networks, why they are problematic, and selfish detection methods that currently exist. It proposes the acknowledgement based method which addresses the shortcomings of existing selfish detection methods. The second paper [2] introduces the selfish awareness dynamic source routing protocol (SA-DSR) which integrates selfish awareness in dynamic source routing (DSR).

DSR is a reactive ad hoc networking protocol that has 2 routing phases; route discovery, and route maintenance. During route discovery, the source node floods the network with route request (RREQ) packets. When each node receives a RREQ packet, they broadcast it to all nodes within reach. Once the RREQ packet reaches the destination node, it will send a route reply packet (RREP) packet back along the same path to the source, including the path it took so that the source can save the route, and use it to send data. Route maintenance is used to ensure that the optimal routes are conserved while nodes are moving, joining or leaving the network. While DSR is an effective protocol for determining the shortest path, it does not consider the possibility of selfish nodes being present.

SA-DSR evaluates the overall selfishness of a path by calculating its successful packet delivery probability (SDP). Each node in the network maintains a “selfishness type” dependent on their battery levels which are: Always Altruistic (AA), Sometimes Selfish (SS), Often Selfish (OS), and Always Selfish (AS). Nodes with low battery levels will be selfish, whereas nodes with higher battery levels will be more altruistic. SDP calculates the average forwarding probability (FP) of all the nodes in the route. FP is calculated as the node’s current battery level combined with its altruism coefficient (AC), a metric used to determine the node’s overall satisfaction with the network. As the source floods the network with RREQ packets, other nodes are expected to retransmit the RREQ packet so that it can reach its destination. If ever another node refuses to retransmit, the source node will increase its AC. For every successful RREQ packet retransmission, the source node will increase its AC. Additionally, the source node will decrease its AC if it does not receive a RREP within a particular time frame. Upon data transmission, the path with the highest SDP will be chosen instead of the shortest path.

Before moving forward, it is important to clarify that there are two types of non-malicious selfish nodes. Pure Selfish Nodes are nodes that display the behaviour discussed in [2], where they take no part in the Route Discovery or Route Maintenance phases. Partial Selfish Nodes, mentioned in [1], are nodes who take part in the Route Discovery and Maintenance phases, but that drop data packets when they are assigned the position of a relay node.

The primary reason that a node would want to become selfish is in order to preserve its own resources, specifically its energy. In most routing protocols for MANETs there is no regulation of the energy consumption distribution across the network. It is entirely possible, even likely that certain nodes will be contributing more of their resources to the benefit of the network than others. The SA-DSR routing protocol proposed in [2] aids in regulating the energy consumption across the network by using selfish behaviour as part of the protocol itself (see above summary for details). SA-DSR uses selfish node behaviour in a controlled manor, removing the need for nodes to become selfish by their own means. On the surface it may seem that SA-DSR is a well-rounded solution to selfish nodes, but there is one core issue; SA-DSR assumes that nodes will only exhibit Pure Selfish behaviour, and only within the constraints of the protocols rules. The protocol does not expect, or address the possibility of nodes in the network exhibiting Partial Selfish Behaviour. This is the problem that we addressed.

To address the issue of existing partially selfish nodes, we have implemented a modified version of SA-DSR by combining it with two selfish detection methods from [1] which are the Acknowledgement based and Reputation based methods. The acknowledgement based method ensures that the destination nodes send acknowledgements to their source nodes to reassure them of a successful packet delivery. The reputation based method checks if nodes are properly co-operating in the network and take note of any signs of selfish behaviors. SA-DSR was modified in this manner to ensure that no data packets get forwarded to selfish nodes.

Modified SA-DSR, functions similarly to SA-DSR. The difference is that it uses TWOACK, an acknowledgement based method, to detect partially selfish nodes. TWOACK states that whenever a node receives a data packet, it sends an acknowledgement two steps back in the communication chain to notify the first node about whether or not the second node participated. Based on this information, the first node will either increase or decrease the route’s SDP accordingly. This is so that if the route turned out to be full of partially selfish nodes, it wouldn’t be selected again in the future.

To determine whether or not modified SA-DSR effectively addressed the shortcomings of DSR and SA-DSR, we collected the following metrics: Average End to End Delay (AEED), Packet Delay Ratio (PDR), Normalized Routing Overhead (NRO) and Battery Depletion Deviation (BDD). AEED represents the speed of which a packet travels from its source to its destination. This is calculated as distance over time. PDR is calculated as the number of packets sent over the number of packets received. This metric is important into informing us of how selfish the network is. NRO is calculated as the number of control packets sent over the number of packets received. This metric is used for the same reasons as PDR. Finally BDD is our newly proposed metric which evaluates the how well balanced energy consumption is across the network.

# Problem and Solution

We have implemented a web application to simulate our modified version of SA-DSR. We have also implemented DSR and SA-DSR for the purpose of comparing them against modified SA-DSR to verify whether it proves to be more effective at discouraging partial and pure selfish nodes in an ad hoc network. Our web application consists of a backend which includes the routing protocol implementation, data collection, logging and exporting. The front end of the application provides a visualization of the ongoing simulation of the three protocols. Our web application is divided into three different sections to for each protocol.

This application was implemented to give us a visual reputation of how the protocols behaves so we can observe how modified SA-DSR discourages pure and partial selfish node behavior. Upon simulation, the application produced data in which we collected, and calculated the average end to end delay (AEED), packet delay ratio (PDR), normalized routing overhead (NRO) and battery depletion deviation (BDD). We fed the calculated data into some data analysation software to obtain a graphical representation of the results.

**Front End**

We decided to implement our very own front end due to the limitations caused by existing applications such as ns-3. We decided not to use ns-3 as the documentation was extremely difficult to obtain through the internet. The easiest way to access the documentation was to obtain a physical hard copy of the documentation in hand. As a result we diverted and created our own front end.

The frontend has a few libraries that it uses in order to make sure that it outputs the data correctly. The main libraries consist of Bootstrap, AngularJS and Highcharts. Bootstrap is a CSS framework that helps in building the UI of web applications - it was used entirely because it is the largest CSS framework out there, the documentation is of outstanding quality and therefore allowed us to easily implement the frontend design quickly and swiftly without any major issues. We used Bootstrap components as well as providing our own styling on the web application. AngularJS is a Javascript framework that allows us to bind data between views and Angular controllers - it played a massive part in our frontend implementation due to its data binding technology it uses. In a nutshell, AngularJS allowed us to update data displayed on the web application without much code. Highcharts is a Javascript library that allows charts and graphs to be displayed on the screen with JSON data provided to it. Instead of exporting our data to a spreadsheet, we decided to implement the graphical data on the frontend to allow quick access to the data straight after running tests with different parameters.

The frontend communicates two ways with the frontend, with a web socket connection and through HTTP calls. The HTTP calls are called using AJAX calls within AngularJS with the main intent of starting simulations. The websocket connection is used to allow real time data to be streamed from the backend to the frontend. We used SignalR to handle the socket connection as SignalR handled a lot of browser type related issues that arise when using standard web sockets, this reduced the code substantially and therefore allowed us to focus on the main aim and functionality of the project. SignalR allowed us to receive node data (and other relevant data) on the fly from the backend. One great example of the SignalR use is where the backend sends requests to the frontend to print transmission arrows on the canvas. The transmission arrows are printed every time the backend requests to do so.

The canvases on each protocol’s tab have been created to have their own history of draw objects such as transmission arrows. The history being kept allowed us to know what was drawn to each canvas and whether or not it should be redrawn onto the canvas or not after canvas changes such as adding new transmission arrows. The history allowed redrawing of states of each simulation whilst the backend sent UI updates in realtime.

The design of the web application has been done in such a way that it would be easy to navigate throughout the different parts such as different protocols, graphs and the output pane. We decided to use tabs due to it allowing us to display a lot of content without having to constantly scroll, which is browser viewport dependant. We also have a input locking mechanism in place to prevent the user from changing parameters that may cause the simulation to produce incorrect data. The relevant input fields and buttons would lock (be disabled) when the user decides to run a simulation. We also have implemented an output pane in the very last tab as a way to show debugging messages to the user whilst they are running a simulation. The output messages update in realtime and therefore, the user can have two sessions open where one browser acts as the output pane and the other acts as the simulation area, allowing direct referencing without having to constantly switch tabs. In addition, a latest output box has been put on the page to show the latest output messages and this also allows the user to see when a simulation has finished, with a sudden flash of green on the border of the latest output box.

All in all, we have provided an elegant alternative to using ns-3 as it has allowed us to demonstrate our project aims effectively by tailoring it to our needs. In the future, it could be possible that parts of the frontend and the backend would be possibly used as a simulation environment for other similar network based research.

**Back End**

The back end of the simulation application is written in C# and communicates with the front end by calling methods in a “MainController” class. All of the work for implementing the protocols is done in the backend, and calls to the MainController are made to log data in the debug messaging pane, populate nodes on the canvas, and print arrows between nodes indicating the transmission of various types of packets.

The network configuration can be randomly generated (as prompted by the front end), by choosing random x and y coordinates within the environment dimensions for each node, and randomly generate pairs of nodes to define the messages to be sent. You can also specify the number of pure or partial selfish nodes you would like to have present in the network, which will set flags in random nodes to direct their future behaviour. The range of the nodes can also be specified, and battery levels between 0 and 100 are randomly assigned to each node.

The battery levels of nodes change by predetermined rates every time it is part of a packet transfer. Whenever a node transmits a packet, its battery level is decreased by 0.02, and whenever a node receives and processes a packet, its battery level is decreased by 0.01. This behaviour is important for calculating the Battery Depletion Deviation for a given test.

The route discovery phases of each protocol involve the “flooding” of RREQ packets to all nodes within range, which in reality would happen concurrently. In our implementation, we are performing this RREQ “flooding” synchronously, meaning that we are using a recursive method to sequentially step through all possible paths that RREQ packets can take to the destination (keeping node transmission ranges in mind), excluding any paths with cycles. Each call of the recursive route discovery method returns a list of all valid routes from that current node to the destination. In the end, the original caller will receive a list of all valid routes, and will be responsible for making a route selection differently depending on the protocol used. The DSR protocol will select the route that has the shortest transmission time. In reality this would be the first RREP received back to the source, but in our synchronous implementation we need to search through all returned routes and calculate the transmission time for each, returning the lowest resulting value. We calculate the transmissions time by using the speed of light constant and the distance between each pair of nodes in the route, where each pixel in the 500x500 simulation area corresponds to 1 meter. The SA-DSR in reality would wait a predetermined amount of time after sending out the RREQ packets, and collect all routes received back until that time runs out. At this point, it would select the route that has the highest SDP value (described in [2] and in the Background section), from the collection of received routes. The MSA-DSR protocol follows a similar methodology as SA-DSR, but the SDP value is calculated differently. As described in the Background section, this protocol tracks the selfish behaviour of neighboring nodes, and incorporates their opinion of these nodes in the SDP of the route. To implement this, given a route, calculate the SDP normally, and then iterate through all nodes in the route, and if one node has a bad opinion about the following node in the chain, multiply the sdp by 0.9. This way the the SDP of a given route is decrease relatively for each potentially misbehaving node in the chain, but will never reach 0. Once the route is selected, we move on to the messaging phase.

The SDP for SA-DSR is calculated in our implementation by taking the average of the Forwarding Probabilities in the route. The Forwarding Probabilities for each node is calculated by averaging its Altruism Coefficient and its “Selfishness Type” decimal values between 0 and 1 which is influenced by the nodes energy level as shown in Figure #1. The Altruism Coefficient is also a decimal value between 0 and 1, and is incremented and decremented based on whether other nodes in the network are forwarding its packets.

|  |  |  |
| --- | --- | --- |
| **Battery Power** | **Selfishness Type** | **Packet Drop Probability** |
| 80-100% | Always Altruistic | 0% |
| 50-80% | Sometimes Selfish | 10% |
| 20-50% | Often Selfish | 50% |
| < 20% | Always Selfish | 100% |

Figure #1: Selfishness Type Table for SA-DSR Routing Protocol

The messaging phase is the same for both DSR and SA-DSR, a DATA packet is sent across the selected route to the destination, at which point an ACK packet is sent back across the route from the destination to the client, and assuming neither of those were dropped in the process, the transmission is labeled as a success. The MSA-DSR protocol has the extra step of performing the TWO-ACK step, where each time a node receives a DATA packet, it sends a ACK packet 2 steps back down the chain to verify that the packet wasn’t dropped by the node 1 step back in the chain. In our implementation, if the packet is dropped, and thus the ACK packets are not sent, the sending node adds the node that dropped the packet to its own list of untrusted nodes which we call the blacklist. This list is used to calculate the SDP for MSA-DSR as mentioned previously.

The simulation allows for the addition of both pure and partial selfish nodes in the network, and the protocols will interact with these nodes according to their behaviour. Whenever a RREQ packet is sent to a pure selfish node, the RREQ packet is dropped, and the recursive route discovery method stops attempting to resolve a route down that path. If a DATA packet is sent to a partial selfish node, the DATA packet is dropped, and the transmission is aborted and marked as a failure. The front end allows you to control how many of each of these types of nodes are present in the environment.

As these protocols run data is collected and stored in a SessionData object for later calculations. Data collected included the number of attempted transmissions, number of successful transmissions, number of control packets sent in the network, end to end delays of each transmission, starting and ending battery levels of all nodes in the network. The SessionData object provides methods to calculate the Packet Delivery Ratio, Average End to End Delay, Normalized Routing Overhead, and Battery Depletion Deviation, all of which are described in the Background section of this report.

# Analysis

While writing the simulation application, there were a few aspects to the protocols that we did not implement as discussed. Firstly, the simulation does not have any mobility. We determined that the mobility of the nodes in the specific test that we are running, has negligible effect of data we are collecting, and yet would drastically increase the complexity of the front end of the application. This would have meant a great deal of time being put into graphics and animation for very little reward. Additionally, we discussed a modified version of the existing TWO-ACK verification scheme which would route acknowledgement packets through an alternate route to previous nodes in the route as a means of reporting selfish behaviour. We did attempt to implement this, but as expected the inefficiency of the scheme made it unuseable. The issue is that the simulation is doing a route discovery phase for each step in the messaging phase, which slows the protocol to a crawl, and makes it extremely difficult to collect good data. Thus, we decided to stick with the existing TWO-ACK scheme which doesn’t add much extra effort to the protocol.

The first simulation scenario is to run all protocols on a generated network configuration with no selfish nodes. The input parameters used were 6 nodes, 5 messages, 300 node range. The results showed that as far as AEED SA-DSR tends to be the best (although is very close to DSR), with MSA-DSR tending to be slower.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.00000215 seconds | 0.00000197 seconds | 0.00000258 seconds |

The Battery Depletion Deviation tends to be the lowest with SA-DSR, with MSA-DSR tending to be pretty close, and DSR lagging behind.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.35 | 0.049 | 0.162 |

The Packet Delivery Ratio averages out to being roughly the same across all protocols.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.7 | 0.6 | 0.6 |

The Normalized Routing Overhead tends to be the lowest with SA-DSR by far, followed by DSR and then MSA-DSR.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 8 | 4 | 13 |

These results support our initial expectations that SA-DSR and MSA-DSR would be a significant improvement with respect to BDD. We were surprised to see however that it also tended to have the lowest NRO. We believe this is because of the RREQ packets that are dropped. This also effects MSA-DSR but that protocol adds additional control packets with each data transmission for the two-ack scheme, and thus ends up with a very high NRO.

The second simulation scenario is when partial selfish nodes are introduced to the network. For these tests, we made 2 out of the 6 nodes partial selfish. The AEED results for this scenario are very constant:

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.00000274 | 0.00000259 | 0.00000262 |

The Battery Depletion Deviation results for this scenario have the same ranking with SA-DSR performing the best, followed by MSA-DSR, and then DSR trailing behind.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.353 | 0.007 | 0.024 |

The Packet Delivery Ratios all drop significantly with partial selfish nodes in the network, with DSR performing the best out of the three protocols:

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.3 | 0.1 | 0.1 |

The Normalized Routing Overhead results are also similar to the previous scenario with the same rankings, SA-DSR performing the best, followed by MSA-DSR, and then DSR.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 67.291 | 24.96 | 36.5 |

The results from this simulation show that SA-DSR continues to outperform DSR and MSA-DSR with respect to BDD and NRO. We also notice that the introduction of partial selfish nodes to the network had a hugely negative effect on all protocols, but DSR managed to win out. It seems that the controlled packet dropping of the SA protocols when in an environment where other packets are also being dropped makes for poor results.

The third simulation scenario is when there are 2 pure selfish nodes and 4 non selfish nodes. The AEED results are once again fairly consistent across protocols with negligible differences:

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.00000102 | 0.00000103 | 0.00000103 |

The BDD results still show SA-DSR as performing the best, this time with DSR performing next best and MSA-DSR performing rather poorly.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.067 | 0.021 | 0.147 |

The PDR results are fairly consistent and better than in the partial selfish node scenario:

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.3 | 0.3 | 0.2 |

The NRO results follow the same pattern as seen previously, with SA-DSR being the best, followed by DSR, and then MSA-DSR:

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 8.5 | 2.7 | 9.75 |

These results indicate that the protocols perform better handling pure selfish nodes than they do partial selfish nodes.

The final simulation scenario has 1 pure selfish, 1 selfish node and 4 non selfish nodes. The AEED values show that in this scenario SA-DSR and MSA-DSR outperformed DSR.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.00000467 | 0.00000206 | 0.00000206 |

The BDD results follow a pattern similar to previous scenarios with SA-DSR beatings out the other protocols:

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.1131 | 0.0054 | 0.1396 |

The PDR results show that the SA protocols can’t manage to complete many transfers due to the loss of too many packets.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 0.3 | 0.1 | 0.1 |

The NRO results once again show that DSR performs the worst, followed by MSA-DSR and then SA-DSR.

|  |  |  |
| --- | --- | --- |
| DSR | SA-DSR | MSA-DSR |
| 86 | 23 | 44 |

All in all the results collected in all scenarios indicate that SA-DSR and MSA-DSR protocols do a very good job of decrease the battery depletion deviation as they are both consistently beating out DSR. Unfortunately this comes with the cost of having worse performance in most other areas when actual selfish nodes are introduced to the network. The design of intentionally dropping packets frequently to conserve battery coupled with malicious nodes in the network means that it is unlikely any transfers complete successfully.

The results also indicate that the NRO cost of MSA-DSR is significantly higher than SA-DSR without much of any other benefits.

# Conclusion

Our simulation application can generate random test data with plenty of input parameters and execute test runs against the well known DSR protocol, the SA-DSR protocol proposed in [2], and our own MSA-DSR protocol. Calculated results from test runs are plotted in graphs in the web application for viewing, and averages are computed and printed as well.

Using this simulation, we have determined that the proposed SA-DSR protocol is very effective at reducing the BDD of the nodes in a MANET, but the cost of doing so is very high with respect to PDR in particular. We have also concluded that our new MSA-DSR algorithm performs slightly worse than SA-DSR with respect to NRO, while not having any major improvements in other metrics.

The goal of SA-DSR and MSA-DSR was to reduce BDD in an effort to remove the desire for nodes to become selfish in the first place. If this has the desired effect on users, and no one becomes selfish on their own accords, then SA-DSR can be viable. If however people still decide to exhibit selfish behaviour despite the new lower BDD, then SA-DSR and MSA-DSR will start to suffer very rapidly.

Overall with some improvements, SA-DSR could be a very useable MANET routing protocol, as long as the low PDR can be resolved.

# References

[1] D. G. Kampitaki and A. A. Economides, "Novel routing protocol for Mobile Ad Hoc Networks with selfish and altruistic nodes," *2016 International Conference on Telecommunications and Multimedia (TEMU)*, Heraklion, 2016, pp. 1-5.

doi: 10.1109/TEMU.2016.7551935

[2] M. M. Ghonge, P. M. Jawandhiya and V. M. Thakare, "Selfish attack detection in mobile Ad hoc networks," *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, Coimbatore, 2017, pp. 1-4.

doi: 10.1109/ICIIECS.2017.8276136

# Packages Used

1. JQuery - Used for simplifying HTML so that it is easier to write
2. AngularJS - Allows Javascript to be executed in the front end
3. Bootstrap - Used for the design of the front end
4. Highcharts - Used for graphing charts to represent the data produced
5. Highcharts-Ng - Used for graphing charts to represent the data produced
6. Angular UI Bootstrap - Used to provide a link between bootstrap and angularJS
7. SignalR - Used to display notifications about the running protocol in the front end
8. ASP.Net - Used develop the web application
9. Newtonsoft JSON - Allowed easy parsing of JSON data and objects within C#

# Setup Instructions

1. Enable Internet Information Services (IIS) for Windows 7
2. Install visual studio 2017 with ASP.NET packages.
3. Download the COMP4203 folder from the submission
4. Open the solution (\*.sln) file called COMP4203.sln under the COMP4203 folder
5. To compile, click on build and select rebuild solution
6. Click the run button
7. Email Imdad Ali, James Tyler or Bradley Koon if there are any issues