Manta: A Peer to Peer File Sharing System for Mobile Ad-hoc Networks

Tanner Haldeman Dhriti Kishore Pia Kochar

Advisor: Boon Thau Loo

Abstract

We created a protocol and system for sharing large media files on mobile devices. Specifically, we sought to solve the issue of being unable to share important visual information in times where there is little to no support from dedicated Internet infrastructure (e.g WiFi and cellular data). Our goal was to develop a scalable and reliable system to support this use. To this end, we worked on developing a protocol named Manta that would permit file transfer transactions to occur in Mobile Ad-Hoc Networks. We then created a file-sharing Android application based on the protocol to demonstrate its practical use and a Python simulator to evaluate the scalability and reliability of our system.

Problem Definition

Mobile devices play a vital role in modern communications. Specifically, sharing media files such as photos and videos allows us to share our experiences with others, transmit important visual information, and document unjust acts in a way that was previously impossible. However, the inconsistency of free cellular data and WiFi coverage limits this mode of communication. To address this problem, we have developed a protocol and system for sharing large files with nearby devices using the WiFi-Direct protocol. Our system relies on peer to peer interaction.

Summary of Solution

The set of Android devices in a specific region form a Mobile Ad-Hoc Network (MANET), consisting of peer-to-peer connections and continuous re-configuration (devices repeatedly forming and breaking connections with one another). This lack of network structure informed many design decisions for our protocol such as using a three-way handshake during a request to ensure files get sent accurately and not storing static information about the state of the network on any node. Our goal was to develop a protocol that scales well to very large networks as well as maintaining fault-tolerance in the presence arbitrary amounts of re-configuration. Therefore, in addition to our Android application, we developed a Python simulation that allows us to alter parameters and collect data on protocol performance.

Detailed Solution Outline

Our solution has three important components. One is that it allows users to upload files to the application from their local devices. The second is that users can list the names of other devices they trust. This is a security related decision, explained in greater depth in the *Ethical, Legal and Social Implications* section. The last, and most significant, is that users can request files by filename from nearby devices that also have the application downloaded.

The upload capability is straightforward. The application shows users a list of files they have on their phone, and they can select which ones they want to be available to others through Manta. This is important because it protects the

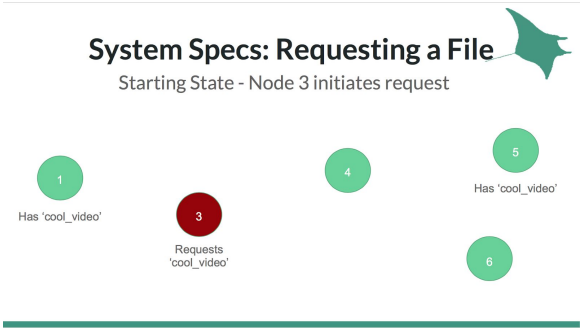
user's privacy by only allowing others to access selected files.

The request scenario is more complex. It is broken into four steps which we call REQUEST, ACK, SEND and FILE. The following table summarizes the purpose of these these steps:

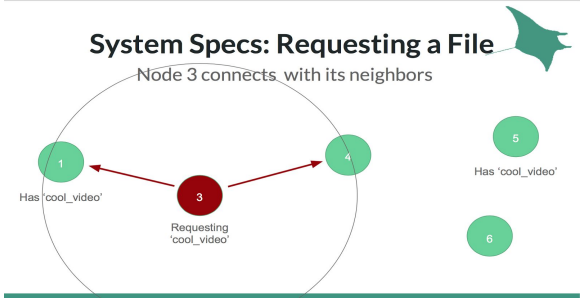
Sent by REQUESTOR DEVICE	Sent by FILE OWNER DEVICE	PURPOSE
REQUEST	ACKNOWLEDGEMENT	Establish path of communication
SEND	FILE	Complete file transfer

The reason for these different stages is to ensure that the only one copy of the file is transferred along one path, and that all the nodes in the network are in agreement on which path this is.

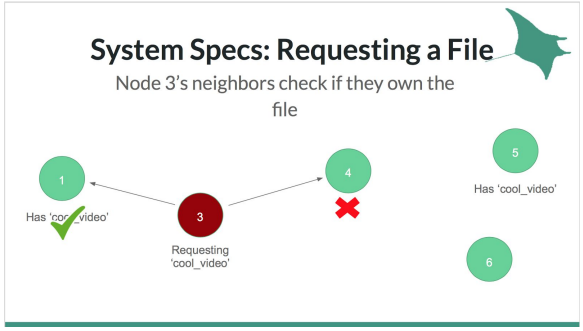
The following graphic shows an example of initial network state. Here, we can see that none of the nodes are connected to each other as they only form connections right before a data transfer. Here, the red node (node 3) is requesting a file from the network. Nodes 1 and 5 both have the requested file. In our system, if multiple copies of a file exist, the one that the request reaches first will send the file back.



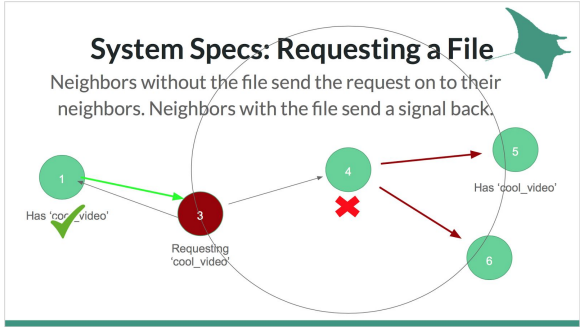
As soon as Node 3 requests the file, we enter the REQUEST phase. In this stage, the original requestor makes a connection with each of the trusted devices in WiFi Direct range and sends a REQUEST packet to each of these devices (shown below).



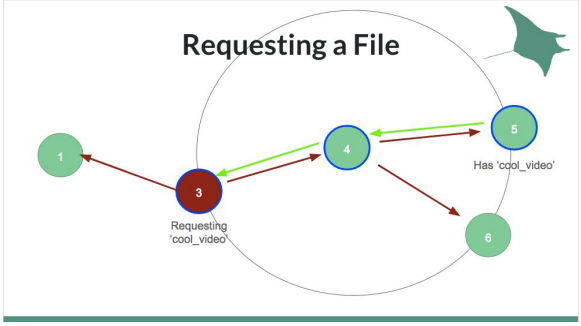
This REQUEST packet contains the requested filename, the requestor's name, the path of nodes taken from the, and the Time to Live (TTL). The TTL is the number of nodes from the current node that we allow the request to broadcast for, to prevent the request from propagating indefinitely or too far. Each of the devices that receive the REQUEST packet then check if they have the requested file (depicted below).



If not, they add themselves to the path, decrement the TTL and re-broadcast the request to all neighbors other than the one the request came from. If a device does have the file, it sends an ACK packet back along the path the request came from (shown below).



If Node 1 hadn't had the file, the following would have happened instead:



As we can see, our system ensures that if the file exists in the network, the requestor will find it via the shortest path possible. Here, we assume that the ACK message received first is from the device closest to the requestor.

The ACK packet contains the same information as the REQUEST packet, except that it also has a path counter keeping track of how far along the path the ACK has travelled.

The request propagates along several different paths in the network, and these different paths have no way of seeing what's happening on other paths. Therefore, if multiple nodes have the file, they will all send ACK packets back to the original node. However, the original node will only proceed with the packet that reaches it first.

As soon as the requestor receives an ACK packet, it sends a SEND packet back through the network along the path recorded in the ACK packet. The purpose of this packet is to notify the closest device with the file that it should send the actual file to the requestor. Additionally, if there were multiple copies of the file in the network, it would prevent multiple devices to begin sending a large file and limiting the capacity of the network. By having this step, we can ensure that the transfer of large file happens over a minimal set of nodes.

Once SEND packet reaches the end of the path (the fileowner), the fileowner sends a FILE packet back along the path. The FILE packet contains the path, the source node's name, a path counter, and the file itself. When this packet reaches the requestor, it stops being propagated and the file is saved to the requestor's device.

Design Decisions

One important design decision we made was that we chose to use unicast rather than broadcast for the ACK, SEND, and FILE messages. We made this decision because unicasting minimizes the number of messages being sent among devices during these transactions, since it reuses the path determined during the REQUEST step. On the other hand, broadcasting in all of these steps would greatly increase the number of in-flight messages to find the file owner and requestor and could overload the network. A downside to unicasting is that it assumes that none of the connections along the path between the requestor and the fileowner will break during the course of the entire transaction. Therefore, our design relies on the network topology remaining stable, because the path determined during the REQUEST step may break in a high churn environment. Broadcasting these messages instead would make the system completely agnostic to the underlying network, and robust in a high churn setting. In order to come to a conclusion, we used information we observed with regards to the transfer rate of WiFi-Direct Protocol. We observed that it was very fast at sending large files, and we hypothesized that if the file transaction took a short time, we would be able to assume a static network topology for that time period and reuse the path of the original request. Additionally, sending more messages through the network could result in the Android batteries draining faster, which would make our application less user friendly. Based on these considerations, we felt unicasting was the more practical decision, so we decided to use unicast messages and focus on shortening the total time of file transaction as much as possible.

Another design decision we had to make early in the process was deciding which peer to

peer protocol to use. Specifically, we had to decide between Bluetooth and WiFi-Direct. We ultimately decided to use Wifi-Direct for multiple reasons. Most importantly, WiFi-Direct has a higher bandwidth and can transmit large files faster. Furthermore, WiFi-Direct has a larger range than Bluetooth, which we thought worked better for our scenarios.

Example of Technical Depth

The technically most difficult part of this project was designing the system and protocol to allow for file sharing over MANETs. An important decision we had to make was what user actions should be captured by the system and how those actions should be proliferated to other devices in the local network. We differentiated between two main designs: an upload based system and a request based system.

In an upload based system, every time a user uploads a file, nearby devices would be notified that the file is now available on this device. The devices receiving this notification would locally store information about what files are available near them in the network. Therefore, devices in this scenario can request files they know to be available near them. A negative aspect of this type of system is that it requires a high volume of in-flight messages to maintain valid copies of available files on each device. Additionally, this system requires a significant amount of local storage, scaled with the number of files available on the network.

On the other hand, in a request-based system, the primary action propelling the system would be a user requesting a file. In this type of system, device would not maintain an index of what files are available on which device, and there would be no notification sent when devices upload files. Instead, users request files and the requests propagate through the network until the requested file is found (or a certain timeout is reached). However, the tradeoff here is that this system requires users to know the name of a file they want to request.

Ultimately, we decided to use the request-based design for several reasons. First, we sought to minimize network congestion by avoiding the constant updates required to maintain lists of available files and paths to those files on each node. A request based system allows us to send only the messages required to complete a file transaction (request, ack, send, file). We felt this was important because these additional messages compete for resources on a mobile device with the messages required to complete file transfers. This would increase the latency and response time for completing a file transaction. In addition to making the application less user friendly, higher latency for file transfers increases the likelihood of the network topology shifting during a transfer and and disrupting it.

Furthermore, the request based system requires less storage, since the upload based system relies on devices storing information about their local networks. Especially in a network with multiple copies of each file, the amount of storage required for the upload based scenario could become prohibitively large for a user friendly Android application. In this type of network, the request based network is much more flexible. In the upload based system, each device would store multiple paths along which to get each duplicate file, and need to execute logic to decide which path to use. In the request based system, however, devices would simply broadcast a request and retrieve the nearest file.

The only drawback of the request based design is that a requestor would need to know the name of the file it would be requesting. While this requirement heavily affects the use cases of our system and application, the decreased latency and storage requirements of the request-based system outweigh this limitation. We therefore decided to develop a request-based system and target scenarios in which such a system would be useful.

System Evaluation

Python Simulator

We sought to measure the performance of our protocol in arbitrary mobile ad hoc networks. As such, we developed a Python application that models a MANET and simulates the behavior of our protocol. Our simulator is highly customizable, and capable of gathering several different kinds of measurements across networks of various shapes and sizes.

The simulator models a variable number of Manta nodes which form pairwise connections when within a certain range of each other. At the start of each simulation step, each node reads any received packets and takes action based on what the message is and who it is intended for (e.g. forward to the next node, reply to the message origin). At each step, nodes can also move within a set distance of their current position. The distance moved, as well as whether to take the move are random and the likelihood a node moves in a given step is a parameter which captures the network's mobility, or "churn."

We decided to measure the file transfer delay of Manta, defined as the amount of time between broadcasting an initial request for a file and receiving the last byte of that file. Because Manta packets were the only messages sent in our simulated network, the protocol's delay was predictably proportionate to the diameter of the network (i.e. the length of the longest path through the network). However, as the topology of this network varies (e.g. because of node mobility), the delay changes in a predictable manner.

We were also interested in the success rate of the protocol. In a static network where all nodes are connected, the protocol behaves deterministically and the average success rate of any file transfer is 1. As node mobility (churn) increases, connections between nodes break and form, and the success rate of Manta transfers drops. This is because several types of packets are unicast between the downloader and uploader, and if the unicast-path is broken before the message is sent, the download will timeout and fail.

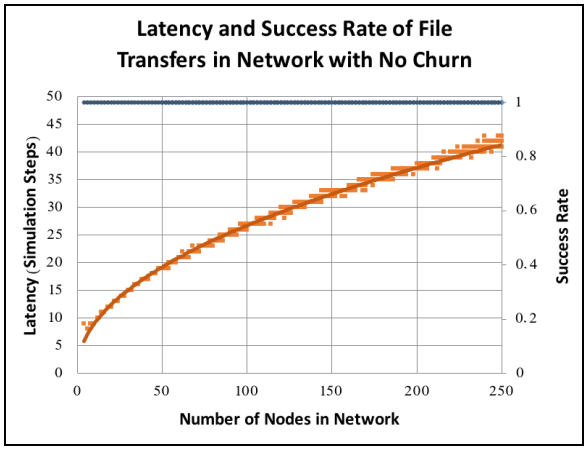
As mentioned, our network simulator is capable of modeling networks with variable conditions. The two independent variables that we chose to focus on are the number of nodes in the network (for measuring scalability) and the mobility of the nodes, or "churn" (for measuring robustness).

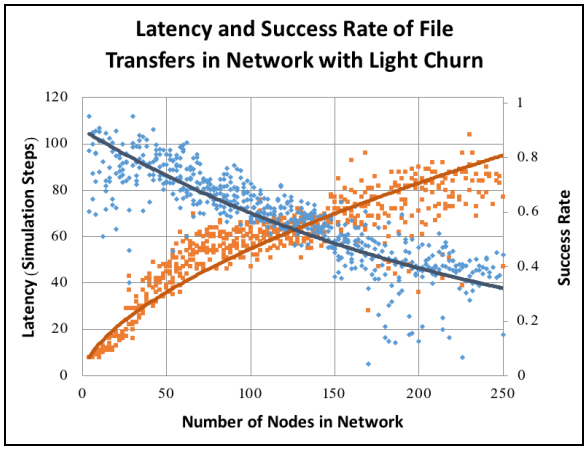
Network churn was set by adjusting a numerical value called the "churn factor" which determines a threshold probability that a node will move during a given step. We chose 3 levels of churn to measure at: "no churn," "light churn," and "heavy churn." The corresponding churn factors were chosen to approximate real physical measurements based on the simulated communication range.

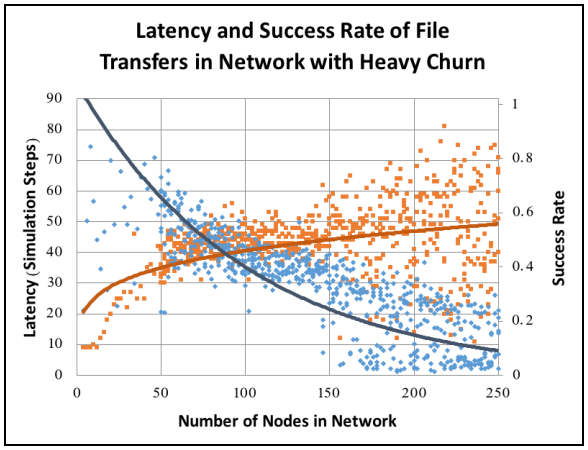
Another interesting parameter is whether to allow nodes to rebroadcast file requests if their download times out, as well as how many rebroadcasts to allow. If rebroadcasts are allowed, file transfer delays may increase in a mobile network as it may require several broadcasts to obtain the file, whereas without rebroadcasts timed-out downloads are considered failed. In our simulation, we allow for 2 rebroadcasts before the download is considered failed.

Simulation Results

We simulated the "no churn," "light churn," and "heavy churn" scenarios described above. We ran about 600 trials per scenario with networks of 4 to 250 nodes. Downloads which timed out after 3 attempts were counted as failures, and the delay of file transfers was measured only for successful transfers. The resultant graphs are below:



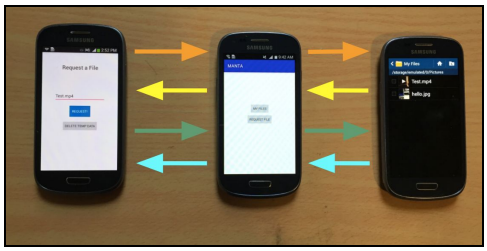




We fitted the delay and success rate for each scenario. The delay was predicted to follow a square-root curve, proportional to the network diameter. The success-rate in the "no churn" scenario was predicted to be the constant 1 because unicast paths are static and cannot be broken. In the "light churn" and "heavy churn" scenarios, the success rate was predicted to follow an inverse exponential curve as the probability that a transfer is successful is proportional that the nodes on the unicast route stay relatively fixed (node movements are independent and identically distributed).

Android Application Metrics

We evaluated the application on functionality, performance, ease of use. The metric we used to measure the performance of the Android application was the total response time of the file downloading on the requestor's phone at varying distances. For ease of use, we qualitatively measured the amount of user interaction required to make the file transfer successful.



Network of devices for collecting metrics

The network over which we evaluated had three Android devices: Device 1, Device 2, and Device 3. Device 1 and Device 3 were not able to directly communicate with each other, but both of them could communicate with Device 2.

The setup of the experiment was that Device 3 had a file that Device 1 wanted to download. Device 1 would initiate a search request for the file and send a REQUEST packet to Device 2. Since Device 2 did not have the file, it forwarded the REQUEST to Device 3. This would be the first step of the file transfer transaction described in the *Solution Outline*. The devices would continue to follow the

protocol specifications to complete the file transfer.

The table below summarizes the total response time for a file transfer over four iterations. One of our goals was to minimize the transaction time and we tried to do so by decreasing the time required to complete the three way handshake. By our fourth iteration, sending a message from Device 1 to Device 3 took an average of 16.67 seconds (over 3 trials) with the same configuration. It would take 50s seconds on average for Device 3 to receive the SEND packet. The final FILE message transfer depends on the file size. For a 0.8 MB file, it took 14 seconds to send the requested file back.

Iteration Number	**Response Time for Request of 0.8MB File**
1	2 min 43 seconds
2	2 min 23 seconds
3	1 min 40 seconds
4 -5	1 min 20 seconds

We also measured the time it took to complete the transaction with 41.67 feet between each device in third floor of the Towne Building. By our fourth iteration, it took 1 minute and 25 seconds. We can see from this the propagation delay at 41 feet did not significantly contribute the response time.

We also attempted to complete the file transaction at 83.33 feet on Towne 3rd floor, in which Device 1 and Device 3 were not in range of each other (could not detect existence of the other Device). However, we found that the Device 1 was having difficulty in connecting to Device 2, and the file transfer could not be completed. This failure was due to the limitations of the Android WiFi Protocol.

Once we were ensure functionality with the 0.8MB file, we also made sure that the file transfer worked for larger files. Specifically, we we were able to complete the file transaction with files of size 7.2MB and 6.8MB. The total response time for the download would only significantly differ on the final FILE Packet transfer (which includes the file being downloaded and depends on the speed of the underlying Android WiFi-Direct Protocol). For illustration, see figure 11.

File Size	**FILE transfer time**
0.8MB	14 seconds
6MB	22 seconds
107.2MB	1 minute 14 seconds

File Transfer Time from Device 3 to Device 1

By our final iteration, we were able to significantly decrease the amount of user interaction required. In our first iteration, there was a lot of coordination required between the users. It would require a high learning curve and not be accessible to users. We were able to simplify the process and make it easier to use by the users. In terms of the trial, the final iteration requires that the Device 2 and Device 3 to be on the system's WiFi-Direct Scanning Page and that Manta application open in the background. Device 1 starts off on the Manta "Request File" page, where a user can input the desired filename. Once the request button is clicked, Device 1 should go to the WiFi-Direct scanning page. With all the devices on the WiFi-Direct scanning page, the file transfer can happen smoothly and quickly. The only button presses required on Device 2 and Device 3 are system dialogs to accept an incoming WiFi Direct connection. The WiFi-Direct specification requires that these dialogs be shown to users.

Ethical, Legal and Social Implications

In any peer-to-peer network, it is difficult to track and protect against illegal behavior. In both peer-to-peer file transfer networks (e.g. BitTorrent) and off-band communication, it is difficult or impossible to prevent the network from being used for illicit purposes such as copyright infringement and the distribution of child pornography.

Since Manta is a completely decentralized peer-to-peer network, it is vulnerable to such unintended uses. It would be possible to monitor traffic across the network by introducing accountability for file transfers (e.g. signing a message with a digital key). An issue with this is that it is unclear to whom these messages would be accountable, and enforcing an accountability measure could substantially affect Manta's robustness and usefulness in many scenarios. Due to the complexity of this issue, we chose to focus on the basic design aspects of Manta in the current iteration of our project.

Another important consideration for this type of network is security. Currently, we maintain a database of trusted peers on each Android device with the application downloaded. This acts as a basic security measure, because the devices only send requests to other devices they trust. However, this does not protect against trusted devices falling into the wrong hands or being corrupted. In the future, it may be desirable to implement a mechanism for dynamic membership to the network, perhaps using an authentication scheme to prevent adversarial connections. Additionally, Manta does not support the encryption of in-flight data or the detection and prevention of forged/corrupted messages, and is vulnerable to many basic network attacks. These problems could be remedied in the future with transport-layer security combined with message authentication codes on each message. Again, due to time constraints, we chose to leave further security considerations to a later iteration.

Conclusion

Our objective for this project was to create a scalable system for sharing large files without the use of dedicated infrastructure. We spent a significant amount of time researching peer to peer networks, and designing our system. Then, we focused on iterating on our application to minimize latency and maximize ease-of-use. This process allowed us to develop the most efficient system we could given time and platform constraints.

Our Python simulation proves that our system design scales to networks with varying size and churn. The system we've developed extends the reach of the WiFi-Direct protocol on Android, allowing users to communicate with devices in a much larger range than they would be able to without our application (in situations with limited network availability).

Future work to optimize this system would include chunking the file into smaller parts and sending them across the network to minimize latency further. Furthermore, we suggest that others experiment with unicasts v. broadcasts to increase the robustness of the system. Lastly, the work we have done can provide a foundation making more complex file sharing systems (e.g. BitTorrent) over MANETs.

Acknowledgements

We would like to thank our advisor Boon Thau Loo for his support and advice throughout our project.