# Armadillo

by

Tural Badirkhanli

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author . [illegible] ...........................
Department of Electrical Engineering and Computer Science
May 26, 2009

Certified by.....................................................
Hari Balakrishnan
Professor
Thesis Supervisor

Accepted by ..... [illegible] ..............
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

# Armadillo

by

Tural Badirkhanli

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

## Abstract

TCP was originally designed to function over static hosts. So, a connection is established between two IP addresses which are assumed to never change of the period of the connection. On the other hand, when TCP is deployed on mobile hosts a number of new factors that are the results of the node's mobility, such as frequent disconnections and changing IP addresses, are introduced into the model. TCP may timeout and quit as a response to these events and therefore yield a suboptimal performance. This work introduces *Armadillo*, a protocol to hide intermittent connectivity from TCP applications on mobile hosts to increase performance. In contrast to all the previous work to our knowledge, our protocol requires no changes to the TCP stack or application on the either end. In a typical scenario we assume that a mobile host uses a WiFi access point (AP) for internet connectivity. Because of the limited range of the AP and the mobility of the host it is going to move out of the range and disconnect. As a consequence, the TCP connection is going to timeout and finally quit. The two important problems we address in this report are the following: (1) preventing the TCP application from timing out and eventually breaking as a result of disconnections and (2) handling the switching between APs so the change of IP addresses is transparent to the TCP application. We evaluate our system under real-world conditions and discuss results.

Thesis Supervisor: Hari Balakrishnan
Title: Professor

# Acknowledgments

First, I want to thank God for giving me patience and strength to complete my work.

I am indebted to my advisor, Hari Balakrishnan, for, first, suggesting this project idea, later guiding me at the times of uncertainty and confusion and coming up with new ideas and even debugging code, and, finally, suggesting this awesome name – Armadillo – for my thesis title. I appreciate all the time you have given me.

I am also indebted to Sivan Toledo who worked with me for long hours be it debugging code or implementing parts of the system and never refused to help. Thank you for all your time, Sivan. I truly appreciate it.

I would like to thank Samuel Madden for working on this project with me and his valuable suggestions.

Finally, I would like to thank Jakob Eriksson, Hariharan Rahul, James Cowling, Szymon Jakubczak, Asfandyar Qureshi, Jayashree Subramanian for never refusing to help.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

TCP was originally designed to function over static hosts. So, a connection is established between two IP addresses which are assumed to never change of the period of the connection. On the other hand, when TCP is deployed on mobile hosts a number of new factors that are the results of the node's mobility, such as frequent disconnections and changing IP addresses, are introduced into the model. TCP may timeout and quit as a response to these events and therefore yield a suboptimal performance.

This work introduces *Armadillo*, a protocol to hide intermittent connectivity from TCP applications on mobile hosts to increase performance. In contrast to all the previous work to our knowledge, our protocol requires no changes to the TCP stack or application on the either end. In a typical scenario we assume that a mobile host uses a WiFi access point (AP) for internet connectivity. Because of the limited range of the AP and the mobility of the host it is going to move out of the range and disconnect. As a consequence, the TCP connection is going to timeout and finally quit. We address this problem in the following paragraph. We use the terms *mobile host* and *client* interchangeably.

In our design, we divide the communication into two parts by introducing two structures -- *proxy client* and *proxy server*. First part is between the *proxy client*, which is running on the mobile host, and the *proxy server* and the second part is between the *proxy server* and the application server. *Proxy server* is a machine operating in between the mobile host (the client) and the server application and is responsible for telling the server to go into persist mode when the mobile host disconnects and otherwise sending the data to and from the mobile host. *Proxy client* is either a process running on the mobile host or a separate machine that the mobile host can connect and is responsible for telling the TCP on the mobile host to go into persist mode in the absense of the internet connectivity and otherwise tunneling the data to the server through the *proxy server.* TCP normally goes into persist mode when the receiver reports a window size of zero. TCP specifications require it to remain in this mode and keep the connection alive until the receiver opens up the window. The two structures in our design structures make it possible to close the receiver window on both sides and prevent them from timing out while the mobile host is disconnected. When the connection comes back *proxy client* and *proxy server* open up the receiver windows on the mobile host and the server respectively and the communication continues. Note that *connecting* means connecting to any AP that has access to the internet.

The two important problems we address in this report are the following: (1) preventing the TCP application from timing out and eventually breaking as a result of disconnection and (2) handling the switching between APs so the change of IP addresses is transparent to the TCP application. We evaluate our system under real-world conditions and discuss the results in chapter 3.

## 1.1 Motivation

Demand for mobile devices is growing strongly as they become more and more popular. Iphone, which constitutes 50% of mobile internet traffic [8], has sold 3.79 million units in the first quarter of 2009, Apple reports [9]. This is the sign indicating the future potential of mobile devices. These mobile devices mostly use 3G celular connectivity for internet communication. However, this service costs tens of dollars a month and is slow. On the other hand, 802.11 protocol was particularly designed for data transfer and is cheaper and faster than 3G. Furthermore, there are a plenty of open 802.11 access points in and around a city that have the internet access enabled and they can be utilized at no cost. A potential problem, however, is that a typical range of an AP is relatively small. A mobile node, once connected to an AP, is soon likely to fall out of range resulting in the connection timing out and eventually quitting. Consequently, the application breaks and needs to be restarted. Note that even if, as a result of falling out of range, the node reconnects to a different AP the application behavior is still the same because the connection that the application was using is no longer alive. The only exception is if the application is specifically designed to operate under such conditions. Obviously, this is not a desirable behavior as it will result in poor performance. *Armadillo* protocol hides disconnections from the application and enables it to run smoothly.

Cabernet shows that there are many open 802.11 access points (APs) around a city that can be used free of charge. In particular, they demonstrate that it is possible to achieve long-term average transfer rate of 38 Mbytes/hour (86 Kbit/sec) [1] while driving in and around a city. This number is promising because it lets the possibility of using a set of TCP applications over open APs and getting a good preformance.

We offer a system that will enable the application to run smoothly despite intermittent connectivity.

## 1.2 Problems with TCP in mobile environments

TCP was originally designed to work on static hosts. Therefore, difficulties arise when the same protocol is adapted to mobile hosts. There are several difficulties the mobility of the host introduces. First, a mobile host is likely to use WiFi for internet communication, which suffers from high error bit rates. Second, the mobility of the host indicates that the host is likely to move and end up in a location with low or no signal at all. This may result in frequent or long term disconnections. Third, the connection is established between two IP addresses that are not expected to change. Changing IP address causes the TCP to break.

## 1.3 Related Work

In this section we discuss related work on the subject.

### 1.3.1 Mobile IP

As mobile host moves from one network to another the corresponding IP address is going to change. This is not a big problem for some stateless protocols such as *HTTP* since two different requests can be made from two different IP addresses. But this is a problem for *SSH* because the TCP connection, on top of which the application is operating, is going to break as a result of changing IP addresses. Mobile IP is a protocol proposed to solve this problem at the network level. The idea behind this protocol is that the mobile host has a permanent IP address, *home address*, where all the data from a fixed host is sent. This data is intercepted by *home agent*, which is located on the same network, using a technique like Proxy ARP (*home agent* advertises a mapping [its own physical address, *home address*]). Consequently, *home agent* tunnels the packets to the *foreign agent*, a router that the mobile host is attached while away from home network, using IP-in-IP encapsulation. *Foreign agent* is responsible for delivering the packets to the mobile host. The routing in the other direction is much simpler – the mobile host simply sends the packets directly to the fixed host and sets the *from* field in the IP header to be *home address*. This routing scheme is often called a *triangle routing*.

Although this protocol hides the changing IP addresses from TCP it is not able to hide the disconnections. As a result, the problem of TCP timing out and eventually quitting persists.

### 1.3.2 M-TCP

M-TCP protocol [7] is a modified version of TCP specifically designed for TCP applications running on a mobile host (MH). M-TCP aims to improve TCP performance for mobile clients, maintain end-to-end TCP semantics, be able to deal with problems caused by lengthy or frequent disconnections. The traffic between MH (the receiver) and the server (the sender) is routed through a Supervisor Host (SH) which hides the disconnections encountered by the MH from the sender and therefore preventing it from timing out and shrinking the congestion window. Modified TCP stack on the MH ensures that the receiver does not timeout and nor shrinks its congestion window. SH hides MH disconnections from the sender by advertising a zero receive window making it appear to the sender that the receiver ran out of receive buffer space.

There is an inconsistency in M-TCP design/implementation. In particular, the authors argue that since a duplicate acknowledgment packet is always ignored the zero window advertisement will be ignored by the sender unless the advertisement acknowledges new data. They address this issue by always keeping the last byte to the sender unacknowledged. If the mobile node disconnects while in the middle of sending ACKs the SH detects this and closes the receive window by sending a zero window packet to the sender and ACKing the byte that was left unacknowledged. But it is not possible to always keep the last byte unacknowledged since if the sender is finished sending and did not receive an ACK for the last byte it will timeout and retransmit that byte. If it nevers recieves the ACK it will eventually quit. This is undesirable and the solution they propose is to send the ACK for the final byte if SH believes the MH will not send any new ACKs. This is justified by that the sender is finished sending and received all the ACKs, M-TCP will not need to close the receive window. However, if the sender decides to send more data and MH happens to be disconnected at that time there will be no unacknowledged byte to close the receive window. According to M-TCP design the SH will not send a zero window packet to the sender in this case and the sender will timeout, retransmit, and eventually quit.

The approach this protocol takes is similar to ours but requires modifications to the client TCP stack and is therefore not very practical.

### 1.3.3 Cabernet

Cabernet [1] is designed to deliver data to and from moving vehicular devices using opportunistic WiFi internet connection. It aims to achieve high data rate throughput despite lengthy disconnections and short timed connections. The authors propose QuickWiFi – a single process that combines all the protocols associated with obtaining connectivity and demonstrate that the time it takes to established connection can be reduced from 12-13 seconds to 400 milliseconds on average. This optimization provides four fold increase in the number of connection opportunities. Cabernet Transport Protocol (CTP) is designed to achieve high throughput given the connection is established.

The experiments demonstrate that %70 of connection opportunities in and around a city last less than 10 seconds and, therefore, given 12 seconds time (on average) required to establish a connection one would be unable to utilize most of the connection opportunities. Once the connection is established the throughput can be decent. The experiments demonstrate the long-term achieved throughput is 38 Mbytes/hour per vehicle (86 Kbit/sec).

Our goals overlap with those of Cabernet in that we both aim to utilize short lived disconnections. We utilize QuickWiFi – a highly efficient technique to establish connections developed by Cabernet – as part of our system. QuickWiFi and *Armadillo* communicates the state of the connection through an interface. QuickWiFi notifies *Armadillo* when the internet connection is gone or reestablished and *Armadillo* closes or opens the receiver window on the client accordingly.

### 1.3.4 Indirect-TCP (I-TCP)

Similar to *Armadillo*, this protocol [4] divides the connection into two parts: one part from the mobile node to the a base station and the second part from the base station to the sender. Base station receives and acknowledges all the packets destined to the mobile node and then takes the responsible of delivering the data to the mobile node. The protocol used between the base station and the mobile node can be TCP or any other protocol optimized for mobile environments. This procol does not provide end-to-end TCP semantics.

### 1.3.5 Freeze-TCP

Freeze-TCP [6] is another modified version of TCP for transmission control on mobile hosts (MHs). Unlike M-TCP, I-TCP, and *Armadillo*, this protocol does not require an intermediary to route packets for it. The MH monitors the signal strength and tries to predict a disconnection or handoff. The prediction needs to be early enough so the MH has a chance to send a zero window packet to the sender and close the receiver window. Once the connection is back the MH sends another packet to open up the window and continue the communication.

It is posssible the MH does not send the zero window packet on time and the sender times out and retransmits. When the MH reconnects it sends 3 ACKs for the last byte received to clear the retransmit timer on the sender and resume the communication fast. Note that it will not be possible to continue communication if the MH reconnects with a different IP address. Also, since the sender times out and retransmits its congestion window may shrink significantly which introduces inefficiency in the performance.

There are a number of factors that make the protocol not fully usable or not very practical. First, Freeze -TCP requires modifications on the TCP stack of the mobile node. Second, this scheme is only useful if the disconnection happens in the middle of data transfer. The authors claim this is the most interesting case. This protocol provides end-to-end semantics.

# Chapter 2

# Proxy System Design and Implementation

## 2.1 Overview

The proxy system consists of two parts – proxy client and proxy server. Proxy client is built on top of QuickWiFi [1]. Proxy client has direct connection to the client machine where the client applications run. and hides the internet disconnectivity from the application client.

The proxy system operates as a middleman between application client and server and prevents the TCP connection from breaking while the client is disconnected from the internet.

## 2.2 Proxy System Design

The proxy system operates in between application client and server and promises to (1) hide the internet disconnection from both ends and (2) handle the changing IP address of the client so the server does not notice it. To hide the internet disconnection from the client and the server the proxy system, when a disconnection is detected and either client or server requests data, replies with a zero window packet to the sender reporting that there is not enough buffer space to receive the data. This technique of hiding the disconnection preserves the congestion window size so the data flow continues at the same rate when the connection comes back.

The proxy system solves the issue of changing client IP address by routing the traffic through a proxy server so that to the application server it looks like the client is running on the proxy server machine. This way the application server always receives the packets from the same IP address - the IP of the proxy server.
In addition to the proxy server which pretends to the application server to be the client there is proxy client. Proxy client sometimes pretends to be the application server and at other times simply forwards through proxy server. A detailed description of proxy client and server follow below.

Note that neither the proxy server nor the proxy client buffer or acknowledge data on behalf of the application client or server. That is, the proxy system does not take any responsibility for delivering data to the either end. This design preserves the end-to-end semantics of TCP.

### 2.2.1 Proxy Client

To obtain access to the internet one or more mobile devices that are running application clients connect to the proxy client which is reponsible for the following: (1) make the disconnection from the internet appear as "insufficient receive buffer space on the server" by reporting a server receive buffer size of zero to the client application and (2) tunnel the traffic from clients through the proxy server and deliver the data that arrives from proxy server back to the clients.

When the application client sends a $SYN$ request to establish a connection to the server, proxy client forwards the $SYN$ request to the server (by tunneling it through proxy server) and immediately replies with $SYN\_ACK$ without waiting for the reply from the server. This ensures that the client establishes the connection right away. At this point, if the internet connection is not available and the client application decides to send data to the server the proxy client immediately replies with a packet indicating a server receive buffer size of zero (WIN0 from now on). As a result, the client aplication goes halt state until an updated window size is reported. In the meanwhile, the client may keep probing the server for more buffer space to which the proxy client replies a WIN0. Once the connection comes back the proxy client reports a non-zero receive buffer space to the application client. The proxy client is built on top of *QuickWiFi* [cite QuickWiFi and describe the interface through a small diagram] which provides an interface to communicate the state of the connection to the internet.

Proxy client encapsulates the IP/TCP packet from a client into another packet and sends it to the proxy server. Similarly, proxy client extracts the original IP/TCP packet from the packet coming back from the proxy server, modifies the destination address and port in the IP and TCP headers and sends it to the client that is expecting it.

### 2.2.2 Proxy Server

IP/TCP packets from application client are tunneled through proxy server which is responsible for the following: (1) extract the original IP/TCP packet from the packet received from the proxy client and send it to the respective application server and deliver the data that arrives from the application server back to the proxy client and (2) make the disconnected client appear as "insufficient receive buffer space on the client" by reporting client receive buffer size of zero to the application server.

Since *QuickWiFi* may acquire different IP addresses as it moves from one location to another and proxy client operates on top of it we need to tunnel traffic through proxy server so that client application data appears to the application server as coming from a single IP address – the IP of the proxy server. The responsibility of the proxy server is to extract the original IP/TCP packet from the encapsulated packet from the proxy client, change the source IP and port in IP and TCP headers and send the packet to the application server. Similarly, for the packets coming from the application server, encapsulate the packet and send it to the last known IP for the proxy client.

If the proxy server does not hear from a particular proxy client for some time $T$ it marks the proxy client as disconnected until it hears from that proxy client again.

When the proxy server receives data from application server for a disconnected client it encapsulates and forwards the data to the last known $IP$ of the client proxy hoping the client will receive it and at the same time reports to the application server a client receive buffer size of zero to prevent the application server from timing out. The proxy server identifies packets coming from a particular proxy client by its unique ID attached to every packet.

### 2.2.3 Potential Problems

We rely on the fact that RFC793 TCP specification does not say anithing about how long the connection is kept alive while the receiver reports window size of zero. We assume this time is long enough so we can tolerate long disconnections. On the other hand, the application on top of TCP may keep its own timer and decide to terminate the connection at any moment. If this happens there is not much that our protocol can do.

## 2.3 System Implementation

### 2.3.1 Proxy Client

Proxy client is build on top of *QuickWiFi* and the state of the connection to the internet is communicated through an interface.

The proxy client keeps a set of variables for every application client in a structure called *app_client*. This structure is created when the proxy client sees a new $SYN$ request from an application client. One of the variables kept in this structure is a *state* with one of the following values:

1. `CLOSED`

*Description*: connection has not yet been requested. This is the default state of the structure when it is first created.

*Entry Action*: None

*Exit Action*: None

*Transition Action*:

**if** [receive *SYN* from application client] **then** [transition to state SYN_RCVD]

1. SYN_RCVD

*Description*: proxy client received a *SYN* request from the application client. *Entry Action*: initialize a new *app_client* structure. Randomly generate a sequence number and reply to the application client with *SYN ACK* with zero window on behalf of the application server.

*Exit Action*: None

*Transition Action*:

**if** [receive *SYN ACK* from application server] **then** [transition to state SAA WAIT]

**if** [receive *SYN ACK ACK* from application client] **then** [transition to state SSA WAIT]

2. SAA_WAIT

*Description*: proxy client received *SYN ACK* from the application server and waiting for *SYN ACK ACK* from the application client.

*Entry Action*: save the difference between the sequence number from the application server and the initially randomly generated sequence number.

*Exit Action*: None

*Transition Action*:

**if** [receive *SYN ACK ACK* from application client] **then** [transition to state ESTABLISHED]

3. SSA_WAIT

*Description*: proxy client received *SYN ACK ACK* from the application client and waiting for *SYN ACK* from the application server.

*Entry Action*: None

*Exit Action*: save the difference between the sequence number from the application server and the initially randomly generated sequence number.

*Transition Action*:

`if` [receive *SYN ACK* from application server] **`then`** [transition to state `ESTABLISHED`]

4. `ESTABLISHED`

*Description*: proxy client received *SYN ACK ACK* from the application client and *SYN ACK* from the application server.

*Entry Action*: Send *SYN ACK ACK* to the application sever.

*Exit Action*: clean the *app_client* structure.

*Transition Action*: `if` [connection is finished] **`then`** [transition to state `CLOSED`]

The state transition diagram is described in Figure 2-1.

If there exists a connection to the internet, every time there is a packet from a particular application client the proxy client looks up the corresponding structure, encapsulates the packet into a UDP packet and sends it to the proxy server. Proxy server, in turn, extracts the packet and forwards it to the application server.

If *QuickWiFi* reports there is no connection to the internet, for every packet with non-zero payload from an application client the proxy client replies with a zero window packet and to the application client it looks like there is not enough buffer space to receive the data. When the connection comes back the proxy client sends a notify packet with non-zero window size to all the application clients it sent zero window packets.

Every *app_client* structure is idenified by a unique *client_number*, which is used as a source port for outgoing UDP packets to the proxy server.

### 2.3.2 Proxy Server

Recall from the design section that proxy server is responsible for (1) extracting the original TCP/IP packet from the client and sending it the application server and encapsulating the reply from the application server into a UDP packet and sending it back to the proxy client and (2) telling the application server to go to persist mode

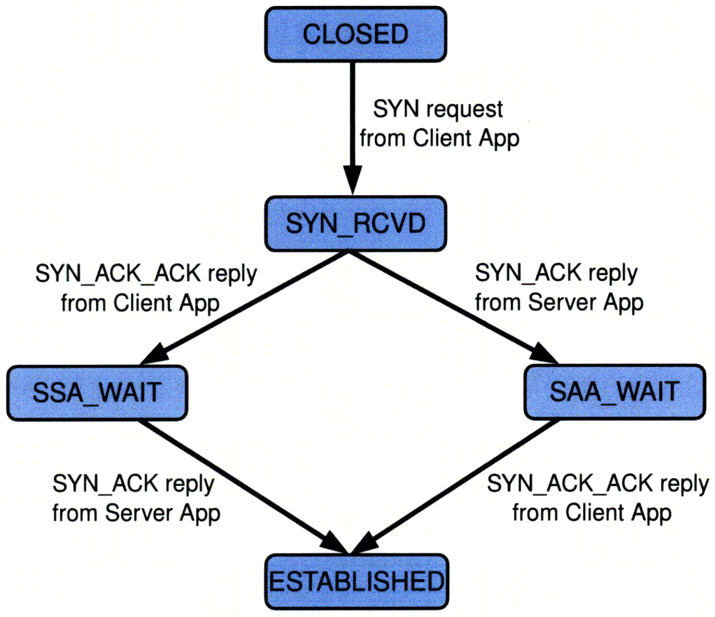
CLOSED
SYN request from Client App
SYN_RCVD
SYN_ACK_ACK reply from Client App
SYN_ACK reply from Server App
SSA_WAIT
SAA_WAIT
SYN_ACK reply from Server App
SYN_ACK_ACK reply from Client App
ESTABLISHED

Figure 2-1: Proxy Client Finate State Machine Diagram

when it detects the application client is disconnected. Similar to the proxy client, proxy server keeps a state for every TCP connection from every mobile host that it is servicing. The finite state machine diagram for proxy server is much simpler than that for the proxy client. The reason is that the proxy server always forwards packets from the client to the server and vice versa except when it detects the mobile host may be disconnected. In this case proxy server still forwards all the packets from the server to the client but also replies to the server with zero window packet telling it to go to persist mode.

Server proxy is implemented in three threads: (1) a thread that is listening on a UDP port, receives packets from mobile hosts and sends them to respective application servers, (2) a thread that is snooping replies from application server, encapsulates the replies into a UDP packets and sends them to the respective client proxy, and (3) a thread that wakes up once in a while to go through all the *app_server* states and delete the ones that have not been used for a long time (10 minutes).

We have a particular design to snoop all the TCP packets that are coming from application servers. We create a special network interface on the proxy machine where all the replies from application servers flow. To acheive this we use a private address to send packet to application servers and use *iptables* to set up a NAT (Network Address Translator) to translate these addresses to public ones on the way out. While packets flow from application server back the proxy server the NAT translates the address back to the private address. We add a routing table rule to tell the kernel to send the packets with that private address to the network interface we set up.

The code is running on a machine connected to the internet through a high bandwidth wired connection.

# Chapter 3

# Evaluation

We perform the experiment described below to show the effectiveness of the system. The results show *Armadillo* is able utilize the short lived connections by achieving decent throughput even when the connection is as short as 10 seconds. [1] shows that over 25% of the connections live at most 10 seconds and there are very few long-lived connections. Therefore, it is important to be able to utilize every connection opportunity. The results show *Armadillo* performs well for short lived as well as long lived connections.

## 3.1 Experiment

In this section we describe the experimental setup and what we aim to achieve.

The experimental setup is as shown on Figure 3-1. An access point (AP) is turned on (given access to the internet) for the number of seconds randomly selected between 1 and 200. Then the same AP is turned off for some time randomly selected between 1 and 200 seconds. The process that is turning the APs on and off are running independently on both access points. The application that is running on the client is *wget* that is fetching *60 MB* file from somewhere on the internet over and over again in a loop. The client machine is connected to the internet through meraki [2] with a wire. The meraki box is running *Armadillo* protocol to connect to the internet. The connection to the internet is going through the two APs, as described in Figure 3-2.

The goal of this experiment is to show that our protocol is able to achieve the goals we discussed in the beginning of this report: (1) prevent the application from timeing out and breaking in the face of short and long disconnection and (2) enable the application to run smoothly (i.e. no need to restart the application) dispite jumping from one AP to another (i.e. changing IP addresses). This is especially important for applications like *wget* or *youtube* as the cost of the restarting the application is too high.

Later, we discuss our experince running *ssh client* or *youtube* through *Armadillo.*

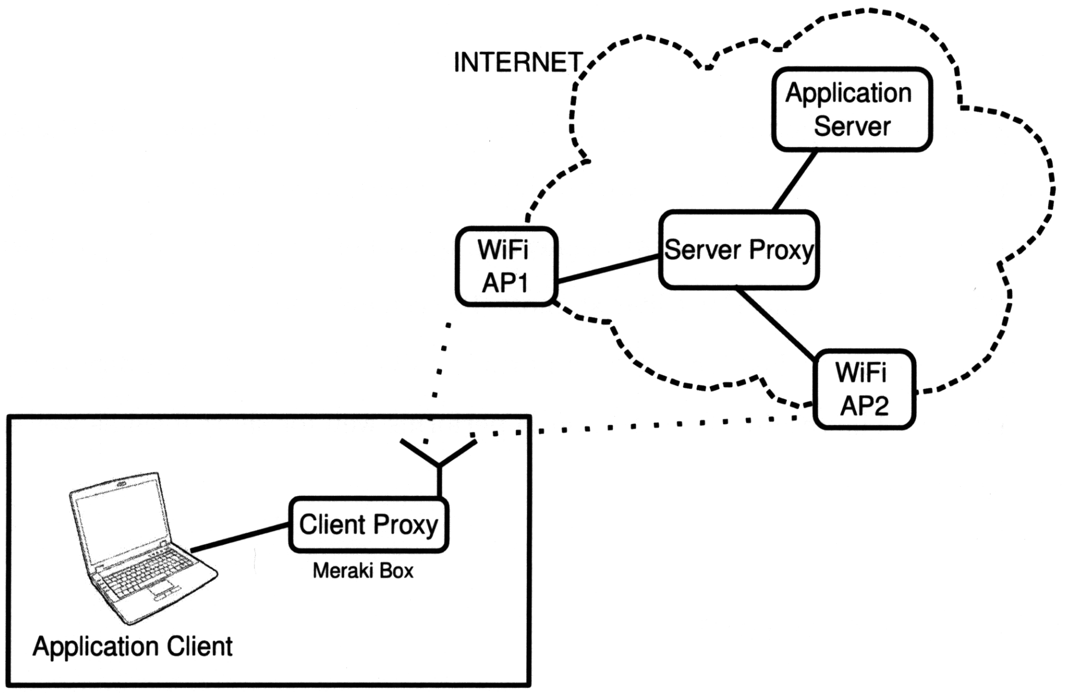

Figure 3-1: Experimental Setup

## 3.2 Results and Analysis

In this section we look at the results from the experiment, analyze them and draw conclusions. Looking at Figure 3-2 we can see a series of short and long disconnections as the application keeps running. The disconnections are as short as 5-10 seconds and as long as ¿ 10 minutes. On average, we are able to achieve 180 KB/s throughput. This is a relatively low throughput compared to what one can get by connecting to an AP directly through their wireless card. We believe the throughput is limited by the processing power of the meraki box. Some implementation specific optimizations are possible to speed up the system. Running the system on a more powerful machine will also yield a better throughput.

Figure 3-3 describes the relation between the connection lengths and the achievable throughput. For connections as short as 10 seconds we are able to get very good throughput value although it is variable. The throughput becomes more stable when connections are longer. The authors in [1] talk about their experience with collecting data by driving around a city and conclude more than 70% of the connections are shorter than 100 seconds. This, once again, shows it is important to stress the importance of being able to utilize the short lived connections and get data through. Our graph shows *Armadillo* is in fact able to utilize short-lived connections to obtain fairly high throughput.

Our experience with using *ssh client* through *Armadillo* is that immediatly switching access points is absolutely transparent to the client. The application continues to run as if the route never changed. When the connection goes away for some longer amount of time the client freezes. Immediatly after the connection resumes the application resume running as well. We had a similar experience with *youtube* videos. Usually the connection is fast enough to allow the buffering of some data. As a result of switching access points the buffering continues with no disruption. If the connection is absent for several minutes and the buffered data is played already the video resumes running right after the connection comes back.

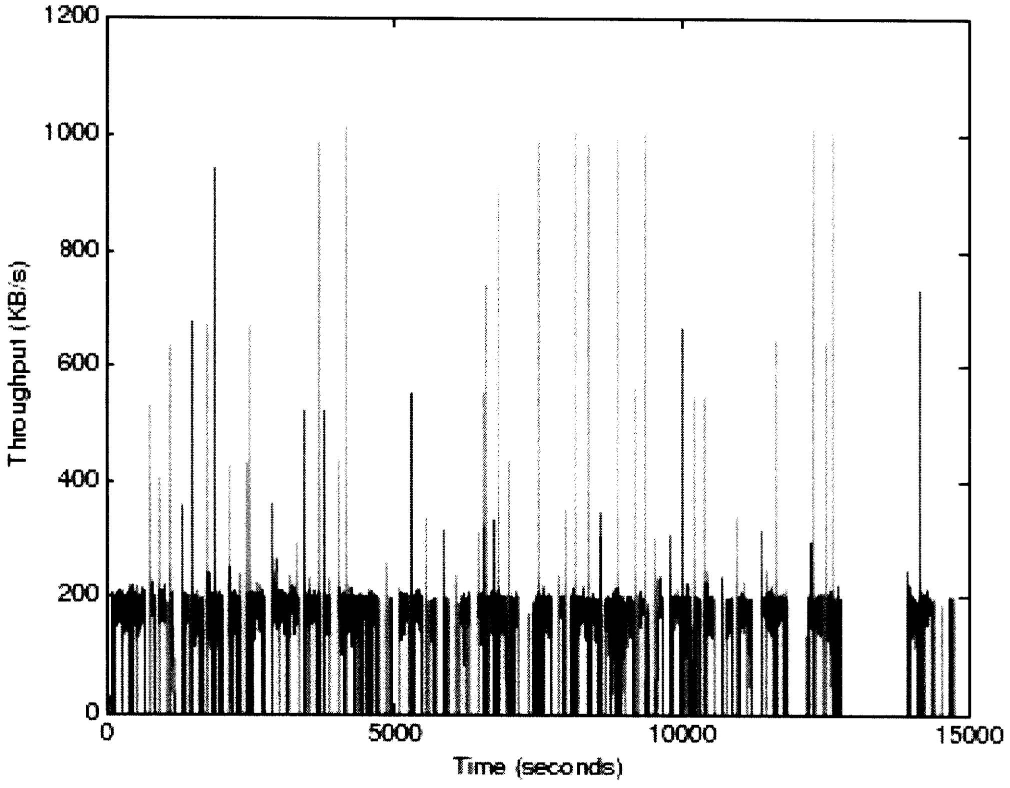

Figure 3-2: Data collected throughout the experiment

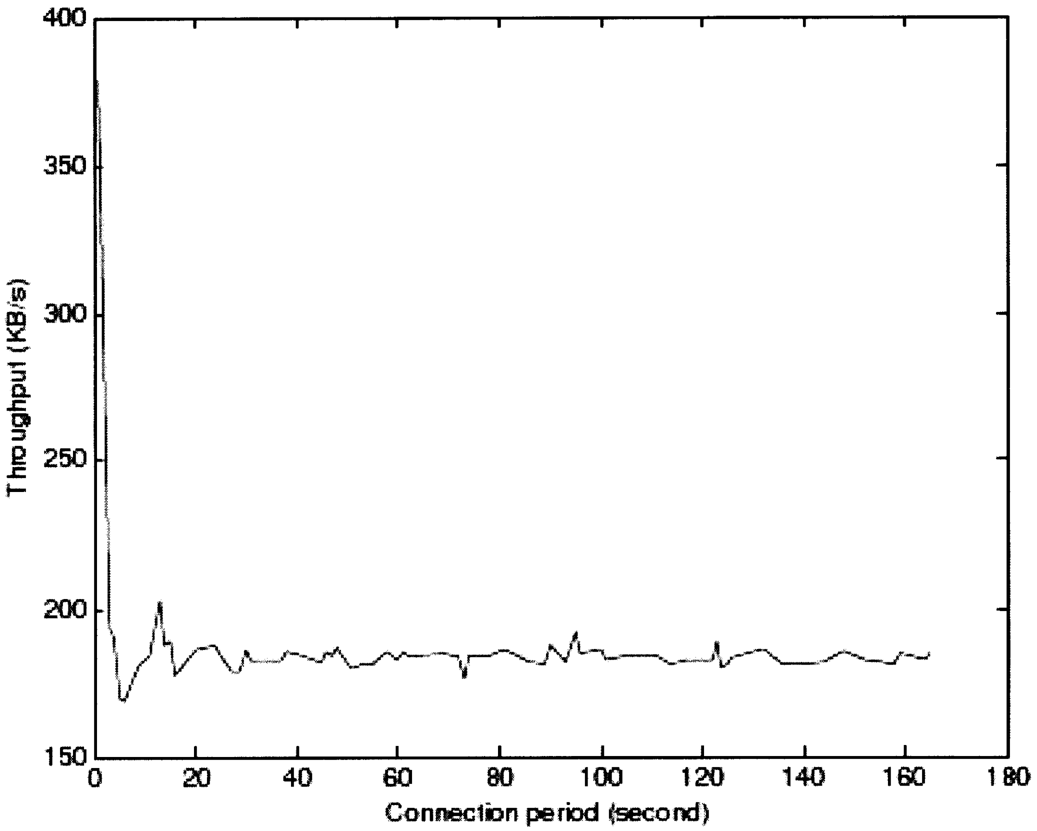

Figure 3-3: The dependency of the Throughput on the Length of the connection

# Chapter 4

# Conclusion

In this chapter we sum up the system design and talk about future work.

In this report we describe *Armadillo* – a protocol to hide intermittent connectivity from TCP applications on mobile hosts. Two main goals *Armadillo* achieves are (1) putting the connection into persist mode in the absence of connectivity and lets the connection stay in that mode until the connectivity resumes and (2) making the changing IP addresses transparent to the application. In chapter 1, we talk about the previous work to our knowledge and discuss how *Armadillo* is different. We discuss the system design and implementation in details in chapter 2. Finally, we show results and analysis in chapter 3.

## 4.1 The System

In this section we talk about the results and conclusions about the system.

The two main parts of the system are *proxy client* and *proxy server*. *Proxy client* is running on the client machine or on a device to which the client has a direct access and is responsible for hiding disconnections and changing IP addresses from the client TCP application. *Proxy server* is located in a fixed location somewhere on the internet and is responsible for hiding disconnections and changing IP addresses of the mobile host from the TCP application server. The traffic is tunneled through the proxy system (i.e. *proxy client* to *proxy server* and back) so the disconnections and changing IP addresses of the client machine can be made transparent to the client TCP application.

## 4.2 Future Work

An extension of this work is for *proxy client* to try to maintain connections to multiple APs at the same time. This has the potential to drammatically increase the throughput of TCO applications [3].

# Bibliography

[1] J. Eriksson, H. Balakrishnan, S. Madden. *Cabernet: Vehicular Content Delivery Using WiFi*, 14th ACM MOBICOM, San Francisco, CA, September 2008

[2] http://meraki.com/

[3] S. Kandula, K. Ching-Ju Lin, T. Badirkhanli, D. Katabi. *FatVAP: Aggregating AP Backhaul Bandwidth*, NSDI 2008

[4] A. Bakre, B.R. Badrinath. *I-TCP: Indirect TCP for Mobile Hosts*

[5] *Routing for Mobile Hosts*, Internetworking, Routing

[6] T. Goff, J. Moronski, D.S. Phatak, V. Gupta. *Freeze-TCP: A true end-to-end TCP enhancement mechanism for mobile environments*, INFOCOM 2000

[7] K. B. And, K. Brown, S. Singh. *TCP for Mobile Cellular Networks*, ACM Computer Communication Review, 1997

[8] http://admob.com/

[9] http://apple.com/