# BAMQP

Dan Robertson

September 2013

**Abstract**

It is often the case in networking that a desired throughput goal exceeded the limit of possible bandwidth. Although bandwidth speed increases 10 fold every few years, newer and faster speeds often come with a price hike that might not be worth the money. Fortunately, there exists a cheap solution that yields desirably higher bandwidth using only existing architecture and a few clever hacks. Link aggregation allows multiple network links to be combined into one logical interface with the combined bandwidth of the supporting links. This has been achieved at the lowest 3 layers of the OSI model, with both proprietary and open software solutions in existence that get the job done.

This MQP goes hand and hand with the ideas and practices behind link aggregation. We will see that while there are many solutions, there is little

## 1 Intro

## 2 Background, Related Work

Related Products:

- AT&T Mobility li LLC holds a patent that applies similar strategies used in this MQP to aggregate WWAN clients who each have a static bandwidth allowance that is rarely in use. At a high level, the patent discusses a strikingly similar approach as this MQP, but is expressed strictly in terms of WWAN. Since this project aims to aggregate WLAN bandwidth, it is not a realization of the invention defined in patent no US7720098 B1 **[Patent US7720098]**.

- Octopus+ is a daemon that allows a computer with multiple means of accessing the internet to dynamically combine the potential links to increase speed. This is most likely a round robin approach between the same internet connections, and the product doesn't do any request splitting. It does gracefully handle link failures and selects the best link for each packet.

The practice of link aggregation was first standardized in the 802.3-2000 IEEE standard. It came as a continuation of the work started by a task force that the 802

group assembled on November of 1997, to address the then disparate trend of link aggregation [802 trunking tutorial]. They recognized that Ethernet was most of the market, and that several vendors already had solutions for aggregating physical links to provide a faster Ethernet connection. However, an inter-operable solution was desired. The protocol they developed became known as the "Link Aggregation Control Protocol"(LACP), officially as 802.3ad, but later moved to 802.1ax [IEEE 802.1ax]. LACP provides a reliable mechanism for aggregating Ethernet links which implement the same protocol. It allows for the controlled addition and removal of links from an aggregation group without disrupting the stream of data frames [IBM IEEE].

One of the core principles behind link aggregation, and this MQP, is cost. Upgrading to a higher speed link is always an option, but trunking together multiple existing lower speed links can satisfy most, if not all, of the same needs. The added bonus is resilience, a failure of one link just means a decrease in throughput, instead of a total loss, as LACP will automatically load balances between the remaining available links. LACP works great for enterprise network operators who have access to the necessary physical links and switches, but has a few short comings. LACP is inherently targeted for neighboring physical connections, which can only be combined through use of additional hardware (namely, a switch). These shortcoming would become more apparent as demand for high bandwidth wireless LAN and WWAN grew.

## 2.1 Heterogeneous Wireless Networks

With the surging popularity of video streaming, high definition content, and online gaming, bandwidth limits are again becoming a problem. Multinode terminal devices (such as smart phones) have the power to actively switch between RATs, but few actively use this technique. Aggregating several Radio Access Technologies (RATs), such as LTE and WiFi, would give these devices better fail over redundancy and increased bandwidth.

Ramaboli et. al present a comprehensive review of the current climate of bandwidth aggregation in heterogeneous wireless networks. The identify a number of common concerns that various bandwidth aggregation techniques must address, in order to be proficient. Packet reordering becomes necessary in any situation when items are delivered out of order. For real time TCP applications, splitting traffic between links with varying conditions such as latency, capacity, and buffer space, results in out of order delivery.[HBA] The second issue, battery power, becomes more of an issue on free roaming mobile devices which aren't constantly tethered to a power source. This MQP effectively sidesteps this issue by performing all aggregation logic on the router.

A tenet of successful bandwidth aggregation is the preservation of packet order. Combining multiple RATs can yield throughput equivalent to the sum of the individual lines. Helps reduce load on a particular link by evenly distributing it over many

links. Distribution order must preserve packet order. Packet reordering does not just exhume buffer space, or slow down reassembly at the client end. Packet reordering outside a certain sequence range will trigger a TCP loss event, which shrinks the transmission window and negatively affect throughput, leading to an underutilization of the aggregate bandwidth capacity.[HBA]

Jayasumana et. al suggest a set of metrics to help evaluate packet reordering, these are Reorder Density (RD) and Reorder Buffer-occupancy Density (RBD). Reorder Density describes the distribution of out of order packets, normalized to the number of packets, for any given sequence. Reorder Buffer-occupancy Density measures the buffer occupancy frequencies normalized to the number of non duplicate packets. This metric can be good for predicting the amount of resources required to preform packet reordering.[RFC5236]

Ramaboli et al conclude that all wireless bandwidth aggregation techniques can be pigeon holed into one of two slots, based on how they deal with packet reordering. Adaptive bandwidth aggregation observes various link conditions in order to smartly route packets of varrying size between interfaces. This would obviously be the prefered method, as it lowers the chances of out of order delivery. Non-adaptive bandwidth aggregation uses static link selection (such as round robin) in order to route sub-flows. This can lead to lower throughput when link and traffic conditions are not constant (as a result of factors such as jitter).[HBA]

## 2.2 A solution for every layer

Bandwidth aggregation has been realized on all network layers except the Physical. These solutions each have their advantages and disadvantages, which are summarized below.

### 2.2.1 Application Layer

Here, the application has access to multiple outgoing interfaces, and can split the data into several application layer chunks (as this MQP attempts to do with HTTP) which it transmits simultaneously over these interfaces. The obvious issue here, is that the aggregation must be application specific, and does not provide more general, application agnostic aggregation. Examples include XFTP, MuniSockets and parallel sockets. Parallel Sockets maintains multiple TCP connections over a single interface to any number of distributed servers, requesting different segments of the same file from each[Parallel TCP]. Multiple TCP protocols (Rome, Paris) leverage the idea of parallel sockets for downloading file portions form distributed servers[TCP paris]. Note that while these methods do not aggregate multiple network interfaces, they do share some common ground with this MQP through their technique of parallel segmented downloading. MuniSockets perform true aggregation of multiple physical interfaces, maintaining a separate thread for each connection.

### 2.2.2 Transport Layer

Multipath TCP (MTCP) has been defined by the IETF in 2011. A Linux Kernel implementation of MPTCP has been developed and is currently undergoing community experimentation, as its popularity grows.[http://multipath-tcp.org/] Apple caused a spike in interest in MPTCP by subtly building it into their iOS7 software.[1] While it is only used for their Siri service in order to provide smooth internet access when one RAT unexpectedly cuts out, it represents the first commercial adoption of the protocol.

Multipath TCP works by breaking outgoing data from the application layer into multiple streams that can travel out different interfaces. The MPTCP stack on the host handles separating each TCP subflow out and gluing returning data from these flows back together, so they can be delivered to the application. Across the network, these subflows are just typical TCP connections, which are handled by each hop as they would normally. This transparency enables MPTCP to work in any situation which traditional TCP would have worked.[MPTCP netys]

It doesn't have support for some of the more desirable bandwidth aggregation techniques, such as intelligent interface selection. To deal with reordering, MPTCP maintains a buffer for out of order packets that are kept until the can be reordered into the stream.

TCP isn't inherently optimized for timely delivery, so using it as a mechanism for aggregation only provides so much gain. Also, bandwidth intensive UDP streaming applications, such as Netflix and Spotify, can't be remedied by MPTCP.

### 2.2.3 Network Layer Solutions

IP protocol is easier to leverage to obtain bandwidth aggregation. Unfortunately, it is prone to more out of sequence arrivals.

The Round Robin packet scheduling algorithm, which runs in O(1) time, is a simple preexisting network layer approach to bandwidth aggregation. Packets (assumed to be of the same size) from the same flow are assigned to multiple paths in a circular matter. The RR scheduler only works so well in theory because it assumes homogeneous packet size and transmission rate. This is very rarely the case however, so the approach falls short.

Kim et al. (2008) proposed a BA scheme that employs two metrics for scheduling, bandwidth estimation and packet partition scheduling. The former determines the amount of bytes that can safely be transmitted across a link without triggering congestion.

[1] Their technical documentation does not advertise their use of MPTCP, but WireShark captures reveal that iOS7 sends MPTCP specific requests in certain scenarios.

Issues with round robin: the bandwidth may be limited to that of the slowest path.

### 2.2.4 Data link Layer

The data link layer was what most of the older solutions to bandwidth aggregation used. It was not until the mobile network boom that attempts to aggregate heterogeneous RATs at the link layer began. One such realization of this practice is the Generic Link Layer (GLL) protocol. GLL is a multi access radio architecture that aims to unite differing RATs under a transparent session. GLL allows for Multi Radio Transmission Diversity. "MRTD is defined as the data flow split (on IP or MAC PDU level) between two communicating entities over more than one RAT. This comprises parallel transmission, or dynamic switching between the available RATs."[**GLL:2005** ] MRTD can select its links based off criteria such as traffic requirements and estimated link quality. However, like many others, this approach to bandwidth aggregation struggles with the issues of packet reordering and unbalanced load distribution.

## 2.3 Adaptive bandwidth aggregation

The pitfalls of the naieve approach (non-adaptive bandwidth aggregation) manifest themselves at every layer. Static decision making for link selection ultimately lead to out of order delivery of differing data segments, which must then be addressed by packet reordering. Adaptive bandwidth aggregation considers the varying link and traffic conditions when organizing sub-flows between multiple interfaces.

Some solutions used at various layers:

- Luo et all (2003) Split traffic into important and non-important stream, delivering each stream over a different RAT. A small training packet is sent along each link to calculate the RTT and delays, this is done intermittently and allows their implementation to better split traffic between different links.

- Splitting traffic into streams is often done with video content, where a base layer (necessary for decoding the video) is separated from different high level enhancement layers, which enrich viewing of the video.

- W-SCTP, an extension of the stream control protocol, which uses a bandwidth aware scheduler (implemented on the sending end) to manage subflows across multiple paths [Casetti and Gaiotto (2004)]. It maintains separate send buffers per interface, and sends each segment down the fastest one until each path has reached its congestion window limit. This approach does not deal with segment reordering, and fails to use the vast majority of metrics for optimal link selection.

- Also see LS-SCTP, which achieves one logical congestion window made up of the aggregate of the participating paths.

- Multipath TCP is a big contender. Arrival time matching load balancing (ATLB) [Hasegawa et al. (2005)], presents a solution to the ordering problem. ATLB scores each path by end-to-end delay, and uses this heuristic (lower is better) to assign each segment to a path. This significantly cuts down on out of order delivery, but a reorder buffer at the client is still maintained to deal with infrequent reordering issues.

# 3 Approach

## 3.1 Big Design Components

**Determining when to aggregate**

- Deciding what traffic to use bandwidth aggregation for. A Naieve approach is to issue an HTTP head request for every packet to get the content size, and judge based on that.
- Since today's web pages often require dozens of resources to be fetched from various locations before a page can be completely displayed, there are many HTTP GET requests associated with a single URL request.
- For a large number of small requests, the round trip delay of sending a HEAD request for every packet introduces an unfavorable latency for typical web sessions.
- Content type cannot be reliably ascertained from just the destination URI or the HTTP headers in the request.
- Content length is specified in the response, so it may be desirable to wait for the response to come back from the server, then canceling the connection and switch to a bandwidth aggregation context

**Negotiating Peer Involvement**

- In order for a peer to participate, it must be within wireless signal range of the initiating router, where the connection between the two is consistent and unlikely to cut out.
- The peer must determine how much of it's total bandwidth it can commit to the session.

**Security and Liability**

- Any traffic downloaded in the name of another peer should be traceable back to that peer.
- Any illegal traffic downloaded using aggregation can potentially be traced back to the owners of each participating router. This risk must be understood by each participating user.

- The traffic downloaded by participating routers must be guaranteed to be from the hosting server, and unmodified. We cannot allow for man in the middle attacks.

**Request Segmentation and Distribution**

- The adaptive solution: when requesting peers for help in a download session, keep a reference of their offered bandwidth. Give this peer a chunksize that is derived by their bandwidth and the size of the file. Ideally, each peer would finish their section in a similar interval.

- Non adaptive: fixed size chunks, assign next chunk to the next available peer until all chunks have been downloaded.

- How do we chose the maximum chunk size? Given the limited space we have on a router, buffer size will enforce this. The buffer cannot be to big, but at the same time, the download speed is adversely affected by request/response time, so less requests for bigger chunks would ensure the greatest download time.

- Downloading by 1 Kb per chunk versus

- Who buffers the responses? If every peer buffers their result and waits until the coordinating router is ready, then we distribute the buffer requirement over all the available routers, which allows for a bigger virtual buffer.

## 3.2 Deciding on a Device

- We knew we needed to pick the right hardware for the job, as this project could be implemented on any variety of devices that can emulate a router.

- The device needed to perform well under normal networking conditions, but needed to be extensible such that device restrictions would not impede the successful implementation of bandwidth aggregation between neighboring nodes.

We narrowed down the scope of our device search to three major families of devices: routers, single-board computers, and Linux computers. The first of these is the Raspberry Pi. Small, cheap, and robust, with a personalized flavor of Linux and support for just about any framework desired, the Pi was an attractive choice. It can be modified to meet a wide variety of needs, one of which includes emulating a router. The pi can supports a growing number of programming languages and boasts 2 GB of free disk space on the cheapest model. However, with a limited number of outgoing ports (2 USB and 1 Ethernet), the quality of router this device could mimic is not so favorable.

Second, a conventional Linux box could have been used. There is a large variety of network purposed Linux distributions and debian derivatives of various capabilities, which could be installed on any Linux platform. This could enable us to use multiple

network interfaces and a powerful CPU with plenty of memory, which could perform well as router. This flexibility makes the combination of a robust distro and powerful hardware a hard option to ignore. The catch however, is the gluttony of the machine. Although the goal of this project is to provide a proof of concept prototype for our model of bandwidth aggregation, a cost effective implementation never hurt anyone.

The third possibility was to use an existing wireless router and flash it with open firmware that would allow it to be more easily altered. The configuration options provided by a commercial router out of the box would not allow for low level modification, and bandwidth aggregation cannot simply be achieved through altering a few NAT tables, so low level modification was a necessity. Because of an oversight of the GNU public license, Linksys developers were force to make their WRT-54G drivers completely public in order to comply with the license. This opened up the door for a variety of hacked WRT firmware to be developed, eventually leading to DD-WRT.

DD-WRT is a buzzword amongst do it yourself networking enthusiast. It provides a large amount of functionality not present on the original router firmware, most of which is exposed in its configuration options. While the added functionality was nice, what we were more concerned with was the ability to load and run custom applications onto the router. For this, OpenWRT (a cousin of DD-WRT) would come in handy.

OpenWRT functions similar to DD-WRT, but it has more developer support and a livelier community then that of the DD-WRT. It comes preloaded with a package manager that makes installing custom software easy, which allows for the creation of C/Python packages and even Kernel Modules. These add-ons can be flashed onto the router as executables, or bundled with the original firmware image.

After reviewing all the device choices, I decided to use OpenWRT for the project. Implementing router to router bandwidth aggregation on a router seemed to be the most straightforward option.

## 3.3 First Design Iteration

After choosing to implement the project on routers flashed with OpenWRT, we began to map out the technical implementation. Our approach would evolve over time as I researched the capabilities and limitations of the chosen platform.

**High level overview of our idea**

1 A daemon will run on each router, that monitors traffic coming through the router, and will decide (based on some selection algorithm) whether or not to perform bandwidth aggregation for the request

2 The router will ask neighboring routers to participate in an aggregation session for the specified request

3 The initiating router will then divide the request into a number of chunks, and hand them to each neighbor.

4 Each neighbor will download the chunk, and forward the response data to the initiating router, who will send it to the client.

Our first hurdle was to figure out how to make the routers exchange data to and from the server, in a way that preserves the router's credentials. For instance, if router A sends a packet to router B that is meant for the server that the host originally requested a file from, B would not know the address of the server, as B only sees the src address from A, and the dest address as its own. We quickly realized that encapsulation was needed.

**Make a diagram of the packet exchange before encapsulation**

Because the two routers need to exchange information freely, we would have to open a TCP connection between both routers. This connection would be set up when a router agreed to participate in an aggregation session. The initiating router would then send request information (URL of server, the size of the file) to the participating router, who would then issue its own request to the destination. The participating router would then set up a mapping, perhaps using Network Address Translation, to forward the responses it gets from the server back to the initiating router, who would then deal with the data accordingly. Eventually, we came to a model similar to this

**Insert diagram of the routers doing their thing**

## 3.4 Implementing on a Proxy

We realized that running a Proxy server on each router would give us the authority to do as much as we wanted with each packet going through the router. For this reason, a number of proxy solutions were researched briefly.

### 3.4.1 Squid Proxy

Squid Proxy, commonly refereed to as Squid Cache, is a caching proxy server for Linux that supports a wide array of features out of the box. It's behaviors are entirely defined by a configuration file that gives a reputable amount of flexibility to the Proxy. Some out of the box features include URL rewriting, request forwarding, dropping, and redirection, content caching, and transparent Proxying.

For the project to be implemented, the proxy would need to be able to inspect a packet, create two sub requests based on that packets headers, and split these requests up between multiple TCP connections. However, since Squid's capabilities are strictly defined by the configuration file, if any functionality wasn't supported out of the box, then Squid's source would have to be modified to accommodate.

Squid enables all of the standard Proxying functions: URL rewriting, content modification, request forwarding, and caching, to name just a few. However, splitting a single request into multiple smaller requests, a necessity for the project, was not

supported. This left us with a choice to either modify the Squid source code, or find another solution. After digging through the source, I found that the documentation was very spotty, and the code was largely unreadable. So although Squid would have been the goto for most Proxy needs, we decided to look for a more script-able solution.

### 3.4.2 Proxying with Python

Python is a popular high level programming language with a meaty standard library that can accomplish both small and large scale needs. With the language focus on ease of use and readability, implementing a Proxy from scratch was both achievable and realistic under our time constraints.

Python has a number of easy to implement http proxy solutions. A convenient list is maintained at: `http://proxies.xhaus.com/python/`. Most of these proxies extend the httplib python library, which provides a simple interface for setting up stable http connections. Python also has a native socket library for TCP/UDP flows, as well as a fair count of other choices for HTTP connections.

At the top of these frameworks, lies Twisted. Twisted is an event driven networking framework for Python. It allows for a fully functioning HTTP Proxy to be created in less then 100 lines of code. Because Twisted is a programming framework, defining new tools and custom functionality is a simple matter of extending the existing tools and combining them in a useful way.

## 3.5 Developing a Prototype

## 3.6 Peer communication protocol

As is typical with most decentralized bandwidth aggregation techniques, involved participants will not only exchange response data, but record keeping and control information needed to keep each peer up to date. I first considered a minimalist approach, in which a coordinating node (who has work to do for one of its clients) queries nearby peers with the URL of the server. If the peers accept, then the coordinator would send requests to each peer continuously, asking for a different bit of the file each time. When a peer had replied with that much data, the chunk was assumed to be delivered, and was passed to the client. I soon realized that this was not enough. There should be a header associated with each message, be it response data or control information. I broke the message exchange down into parts, and began to postulate what sort of information would be needed for the headers of each (worrying about the actual implementation of the header later).

**Session initialization:**

```
A request sent by a coordinator who wishes to begin an aggregation session
with neighbors.  This is usually broadcasted to nearby peers.
```

- All information necessary for a peer to connect to the host. This would be the URI, protocol, and port at a bare minimum.
- Size of the file. A peer should know the size of the file to be downloaded ahead of time, so it can estimate how much time the session would require of it, and whether that would negatively impact the QoS of its clients.
- Identification information, that identifies the source of the request. This could be implemented through cryptography by encrypting the URL or some part of the request with a key specific to each router. When a neighbor gets a request, they will attempt to decrypt using that peers public key. If the decryption is invalid, then something is amok.

**Accept/Decline session:** `An Accept/Decline message is sent by a peer who wishes to participate in a session, or to be left out (respectively)`

- For accept, the peer should include the original URL, as well as a code indicating that they ACCEPT the session.
- An ACCEPT message should optionally respond with the instantaneous bandwidth that the peer can promise.
- A DECLINE message should respond with a reason, such as "too busy" or "unwilling to work for untrusted peer".

**DROP session** `Sent out by a peer who wishes to break out of a session while it is in progress.`

- The request should also give a reason as to why the peer wishes to terminate.
- The request may optionally tell the coordinator to remove them from their peer list for a given amount of time.

**Chunk request:** `A chunk message requests a specific byte range from the target resource. This message is passed to a peer who is then expected to either retrieve the data and pass it back to the coordinator, or respond with a drop request`

- As stated, the range should be indicated.
- the response from the peer should indicate the range that it downloaded, so that the information can be properly reassembled before being delivered to the client.

After the logic behind the protocol was outlined, I still had to figure out how to implement it. My first approach was to simply pass the header values with CR/LF delimiters, and parse them myself. However, it occurred to me that a custom header would involve more work as the protocol grew in complexity (something which I

wished to allow for).

Since the framework I was currently using was designed around the HTTP protocol, and provided a plethora of functions for it, I considered using this as an underlying protocol for my bandwidth aggregation protocol. The beauty is that HTTP provides a number of useful headers that translate well to the need outlined above. I decided that this would in fact be the proper protocol to use, as it is relatively lightweight, but still powerful and verbose enough to articulate the needs of each peer during any arbitrary interaction. What's more is that HTTP allows for non standard headers to be included. Any subset of ASCII characters can be included as a header name, which would allow us to encode any information into the headers.

Implementing HTTP interactions triggered a major code overhaul, as ideas were more clearly defined, and implementation details were fine tuned in accommodation. I realized that when working for a coordinating router, peers act in an isolated request/response manner, rather then a persistent open stream that need not follow the request implies response architecture. When since the coordinating router is in charge of dispatching work for particular file portions to each peer, the involved peer only needs to know two things: what to get, and where to get it from. With HTTP, I was able to seperate each request that a coordinator would send to a peer, and map it to its own url. The outcome is somewhat of a lightweight API running on each peer. Each message type is bound to a URL that the peer is listening to. It fetches the resource (if one is requested) for the coordinator and pipes the response back over a persistent TCP connection opened up when the coordinator hits the /INIT url. This connection can be closed by requesting the /DROP url. The HTTP implementation allows for the interactions with each peer to be very clear and well known. The URL format follows that of [protocol]://[IP]:[port]/[REQUEST TYPE].

# 4 Evaluation

# 5 Discussion

## 5.1 Rationale for various Design decisions (and their alternatives)

### 5.1.1 Packet Segmentation Algorithm

The naieve approach, was fairly straight forward. Given a peer network of n routers, give each router:

$$\frac{FILESIZE}{n} bytes$$

This has a major shortcoming however: different routers download at different rates, so the file download isn't complete until the slowest router has downloaded and trans-

mited its chunk back to the host. So the download time becomes the download speed of the slowest router. A better approach is to divy up the file into chunks using a relation between individual router bandwidth and total pool bandwidth.

A consideration to keep in mind is that the host router may not be directly connected to each available peer, and certain peers may be connected to each other better then the host router. It could be the case that the host has 3 peers, who each have 3 other peers in wireless communication range. When the router divies up these chunks, the advertised bandwidth of a router that he is directly connected too should reflect the average bandwidth of all of that routers peers. So if A connects to B, who has 2mbps of bandwidth, but B can talk to C and D, who each have a 10mbps connection, B should advertise to A that his bandwidth is

$$\frac{B + C + D}{3}$$

mbps. This will help A more accurately decide how to manage the file segmenting. When B is passed this chunk, he can do the same with each peer in his network.

The problem of overlapping neighbors does immediately become an issue. I plan to address this at the implementation level, and assume without loss of generality that this can be accomplished. My plan so far is to have the host generate a random session key, and pass the key along when it communicates with its neighbors during the negotiation period. Each neighbor will store this key, and until a cancel request is sent from the host, the router will reject any negotiationg requests whose key matches their stored key. This way, a router will not commit its bandwidth too two different neighbors on the same file download.

The second issue might be link cost. Presumably, routers will communicate with each other over 802.11 b/g at 54 mbps. However, this link cost is negligible, as to reach a point of diminishing returns would take an unusual amount of hops.

Pseudo code of the algorithm:

**for all** peer in neighbors(host) **do**
  $netBandwidth \leftarrow netBandwidth + peer.bandwidth$
**end for**
**for all** peer in neighbors(host) **do**
  $Chunk_{peer} \leftarrow \frac{peer.bandwidth}{netBandwidth} \times fileSize$
**end for**
**for all** peer in neighbors(host) **do** Delegate $Chunk_{peer}$
  $AmountRemaining \leftarrow AmountRemaining - Chunk_{peer}$
**end for**
Issue HTTP GET for $AmountRemaining$

### 5.1.2 Boundaries on segment size

There are a number of problematic pitfalls that this type of segmentation produces. When a file is downloaded in one session under normal conditions, the round trip time for the request (reaching the server, getting the first response) has a negligible impact. As TCP's sawtooth pattern kicks in, an appropriate window size is established and transmission time smooths out. The TCP overhead can be avoided because HTTP 1.1 uses persistent TCP connections, so the setup only has to happen once for each peer, just as with a normal download.

The round trip delay (RTT), while negligible for single session downloads, presents an issue when the download is broken into segments. The RTT overhead is applied to each segment request, for each peer. This gives the following relation:

$$Overhead = \frac{filesize}{segmentsize} * RTT$$

The overhead grows as segment size decreases, so a larger segment is desirable. This presents a problem for devices with limited buffer space. For example, a typical home router usually has only a few kilobytes of total buffer space. To use all the available buffer would result in buffer bloat, leading to congestion at the link, so a fraction, no bigger then perhaps half the available space, should be used. This means that each router can only download one chunk at a time, and must wait until the client is ready to request more. This idle time should also be considered.

Fortunately, the overhead is outweighed by the gains of downloading in parallel.

## 5.2 Security Goals

The biggest problem with our approach to bandwidth aggregation, is that it puts the end client in control. The two big concerns are data integrity and download liability. The former will be achieved through an implementation of the zero knowledge protocol, the latter will be addressed by asymmetric cryptography. In addition, the goals of Availability and Confidentiality will be considered. Availability is by far the most important factor, as a disabled router will certainly lead to quality of service concerns on the client end. It is one thing to guarantee faster internet, but to do so at the cost of losing access entirely is unacceptable. Confidentiality is certainly a desired, but difficult to achieve in our case, as the operator of a peer's router has the right and power to view any traffic allocated to it.

The easiest scenario to envision for accessibility issues is that of a denial of service, instigated by a peer. If a peer could simply demand bandwidth from its neighbors endlessly until they were no longer capable of serving their own clients, then accessibility is certainly breached. For this reason, peer routers volunteer at will to join aggregation sessions. If a client on a peer's network suddenly demands more bandwidth, the peer may opt out of the session, and the initiating router will react accordingly. The routers will listen on a shared TCP port, which any peer may close themselves if they wish to opt out of bandwidth aggregation (in either direction)

Blindly accepting file data from a peer router would certainly be a huge security hole, so the issue has to be addressed. In cryptography, the Zero Knowledge Proof is a technique in which two parties, a prover and a verifier, exchange and verify the truth of a piece of data without the use of external information. In the case of this MQP, the client (the verifier) must check segment data coming back from each peer (the prover) for validity. To do so, the client will choose a random section of bytes within the range of the segment range it asked the peer for. When the peer responds, it will check the response against the verify bytes that it downloaded its self, rejecting the response if the bytes don't match. A malignant peer has no way of knowing which segment the client will check, so short of a very lucky guess[2], the peer has no choice but to send the legitimate segment. This is both a secure and easy to implement solution, with only a small degree of additional overhead (the RTT for each segment is now doubled, as two requests must be made to the server). The overhead is a negligible price to pay in the grand scope of data integrity.

### 5.2.1 Trust Platform, and Reliability

It is important in our aggregation model, that participating routers have a well founded trust established between each other. A peer that is caught sending maleficent data should be subject to significantly more skepticism then a peer who has no violations. Similarly, a peer may not want to involve its self in a session that an untrusted neighbor is initiating. A security trust relationship is needed, and it must be dynamic and responsive. Selecting peers with desirable resources (bandwidth) and trust can be gone about in a variety of ways, which makes this a matter of picking the right approach for the job. While in certain scenarios, a trusted third party could alleviate some or all of the pressure involving trust calculations, we are unable to make use of it. The trust model must be dynamic and self regulating. A reputation based model could be used, where peers seek information about the quality of past experiences other peers have had with a untrusted peer, evolving into a semi-complex who-trusts-who social network.

The definition of trust: Trust is a quantified belief hold by a peer, which is formed through the observations and recommendations, with respect to another peers ability to complete a particular task successfully.

The Peer to Peer (P2P) network model has all these demands. I looked to this field for answers and insights into how I might go about developing a trust model for my protocol. As it turns out, this is an open area of research in P2P, so there was a great collection of data to check over. One such model employs data-signing in order to verify how credible data coming back is. Peers who have more valid data-signatures under their belt are deemed more trust worth. This is a popular approach to trust in file sharing P2P applications. In larger P2P networks, a single client can't possibly interact with every other node its self in order to evaluate trust. This puts forth the

[2]With a segment size of 1000 bytes, and a verification size of 4 consecutive bytes, the probability of both the verifier and prover choosing the same sequence is $2.006 \times 10^{-9}$

need for a reputation based trust model, where a peer evaluates the trust level of another peer through the claims of other peers who have interacted with the peer in question. Xiong et al identify a number of other concerning characteristics of the reputation model. For instance, if the feedback mechanism is not incentivized, then a peer being asked for information regarding its experience with another peer may simply lie, and poison the askers understanding of the peer in question. A peer may also discretely raise and lower trust by preforming many small transactions truthfully, in order to build sufficient trust to be involved in a larger session, which it may then lie about (a sting operation!)[Xiong PeerTrust].

Xiong et al introduce an elaborate approach that attempts to fix most of the outlined problems with a reputation based model. This includes feedback scope, which adds a context of the feedback (was the interaction trivial or monumental), as well as credibility factor for each peer, which is used to asses the trustworthiness of a feedback provider[Xiong]. For our application, the reputation model may not be necessary. In a practical sense of our application, there will likely be less then 10, certainly less then 100 peer nodes in a single wireless bandwidth pool. Further, since each peer is within a somewhat close proximity, and each peer is maintained by a real individual (the router owner), it is less likely that either a peer or its owner will never interact with another peer. In our case, the overhead and unreliability of the reputation model would not be worthwhile. Where we to do one, it would likely work as follows: at the end of an aggregation session, the coordinating peer would broadcast its peer feedback report to all peers involved in the session, so participants would gain a better understanding of other participants who they were not directly involved with. The benefit here is that a black sheep can be identified immediately. However, the downside is that the coordinating peer has too much power. It can easily lie about any interaction it had.

Accumulating trust through record keeping is a reliable method, but falls short because due to its slow start methodology[p2pTrustRecommendation].Our bandwidth aggregation schema is more static, in that neighboring peers who involve themselves are likely to be in a relationship for a long time. A real world example to relate would be that of a new neighbor moving into an apartment complex. Suppose that some or all of the residents in this complex are using the bandwidth aggregation model described herein. If the new neighbor wishes to join in, the other routers will at first be wary, possibly choosing to include him in a only a subset of interactions. But once trust has been built, it will pervade until the peer performs a malignant action, or the peer‘’ router's owner moves away. Since new nodes aren't continuously swapped in and out on a daily basis, a local trust model should suffice for our application.

But how could these trust calculations be modeled mathematically. One approach, suggested by Medic et al, is to represent a peer's trust factor as a tuple of Trust and untrust. Mathematically $F = [T, U]$ Where $T = Trust$, and $U = Untrust$. The benefit of this model is that differing weights can be assigned to each trust factor component. This simulates a weighted average of the two factors.

In our protocol, it may be the case that a peer who misbehaves in the past corrects themselves for all future interactions. Their past violation should only act as a brief hiccup, and not hurt their trust factor for the rest of their existence. This desire suggests a weighted average of each trust factor, between past and recent interactions. Past interactions would be weighed less heavily when computing a peers trust factor, whereas recent (within the past 12 hours) would be assigned a heavier weight. This allows for a peer's trust to evolve with their actions, and scale off better recent behavior. Each trust value will be within the interval of [0,1]

Borrowing from Xiong et al's approach to trust computation, we chose to store trust on a transaction level. Here, a transaction represents one download session held between a coordinator and a peer. The transaction will hold the computed trust factor for that session. If all goes well, the value will be [1,0]. However, if the peer made any untrustworthy actions, the second value (untrust) will be higher, resulting in a lower trust value. The trust factor metric is computed by the summation of each of these transactions (with a higher weight assigned to more recent transactions, good or bad). In their PeerTrust model, Xiong et al model feedback that the coordinator has with peer $u$ from transaction $i$ as $S(u,i)$. We will use this normalized amount of satisfaction for the formation of our trust factor score. Note that $I(u)$ denotes the number of transactions the coordinator has had with peer $u$, $W(u,i)$ denotes the weight assigned to the feedback the coordinator has had with peer $u$ at transaction $i$ (based off the time of transaction), and $R(u,i)$ denotes the untrust feedback from transaction $i$ with peer $u$. With this information, $T(u)$ (trust) and $U(u)$ (untrust) are calculated as follows:

$$T(u) = \frac{\sum_{i=1}^{I(u)} S(u,i) * W(u,i)}{I(u)}, U(u) = \frac{\sum_{i=1}^{I(u)} S(u,i) * W(u,i)}{I(u)}$$

Once the metric has been computed, the trust making logic is fairly simple. If $T(u) > U(u)$, then the peer is trustworthy. If not, the peer cannot be trusted. Since trust is just one of the many metrics that will go into peer selection, simply assessing trust on a binary level is sufficient.

Given this formula for averaging trust, we still must add one more vital component before a score can be derived. The trust value its self."Trust is fuzzy since trust is imprecise and vague. Trust is dynamic since it is not stable and it changes as time goes by. Trust is also complex since different ways are possible for determining trust."**[Chang, E., Dillon, T., Hussain, F.K.: Trust and Reputation for Service-Oriented Environments.Wiley (2006)]**.

In digging through multiple papers on peer2peer trust analysis, I found a few terms and considerations that need be introduced. With unforgeable identities (in our case, private keys generated by a trusted central source), it should be difficult for a peer to erase their given identity (and reputation associated with it) and begin anew with a fresh identity. Such a peer could use this technique, known as whitewashing, in order to rebuild their reputation. This is an issue in systems where new peers are trusted from the start.

Stakhanova et al present a through outline of Trust information, in terms of what is gathered and how it is scored. The transaction quality can be modeled as positive, negative, or a combination of both. They note that a versatile will consider both, as relying on only negative feedback can may lead to wrongful elimination, while relying on only positive can allow malicious peers to fake benevolence. Second, the size of the files exchanged can also play a role.[Stakhanova]

Trustworthiness can be modeled in multiple ways, where the chosen method is entirely dependent on the needs of the system. Some P2P systems, such as XREP and Travos, store trust as a binary value, meaning that the transaction was or wasn't satisfactory. The converse approach is to model trust as a discrete value on the scale of [0,1], allowing for partial trust to be modeled[Stakhanova]. In many systems, it is important to record the time of the interaction. This allows recency to be factored into trust decision making, like giving less weight to older values.

Different approaches to transaction storage will yield different trust values. The memory efficient approach would be to store one average record, and update each component on a per transaciton basis. This has great appeal on devices with low memory (such as routers), but it sacrifices the elements of recency that help contribute to a more accurate understanding of the peer's trust. However, storage requirements in this scenario increase linearly over time as the number of peers in the system grow.

Stakhanova et al note that in cases where a few well known peers interact with each other often, storing trust locally is sufficient for each of them to make decisions. This obvioulsy doesn't scale for applications with millions of participants. This project will adopt the former approach, as the number of involved peers are limited to a finite local wireless range, which wouldn't typically exceed more then a dozen peers[Stakhanova].

How do we compute the actual trust calculation? The smallest scale we can start on, is at the per-chunk level. Selcuk et al present a solution that uses a bit-vector of m interactions. The bits are either 1 or 0, when the data is authentic or inauthentic, respectively. For our project, each peer has a chance to verify every chunk that is sent. The bit vector could hold $m$ bits, where $m$ is the number of chunks verified. As trust grew, $m$ could become smaller and smaller, reducing the verification overhead. The overal trust from a transaction is computed by treating the $m$ significant bits in the vector as a signed integer[Selcuk].

$$\frac{m - bitvector}{2^m}$$

Dividing by $2^m$ produces a score on the interval of [0,1). This score can then be stored in the transaction record, and used in the averaging function mentioned above. The benefit of the system is that it accumulates a continuous trust value from many binary transactions. When a file is verified, it either passes verification or fails.

### 5.2.2 Responsibility with Asymmetric Cryptography

Since bandwidth aggregation is often used as a means to accelerate download speeds for large files, some considerations regarding these very types of files arise. Large files

can be many things, such as a computer application, a compressed music library, or video data. All these cases are susceptible to copy right infringement due to illegal download. This scenario introduces a key problem that concerns all the peers. If a client wishes to download copyrighted material illegally, and aggregation is used, each peer router owner is liable for the infringement, not just the client. This is explained in the legality section of this paper. Since it is impossible for each peer to asses the legality of a file being downloaded, there is little that a peer can do to prevent themselves from incriminating their clients (the network owners).

If a peer router maintained a log of each session that it participated in, which could definitively map each request made back to the coordinating host, then the peer could deny their liability in the implication, at least to some extent.[3]. If the log file could be falsified or changed, then it wouldn't hold up very well. Thankfully, there is a way to map each request back to its originator: Asymmetric cryptography. Commonly referred to as public key cryptography, this cryptographic algorithm works by using two keys or signatures to encrypt or decrypt a piece of data. One key is made public, while the other is kept private. These two keys are intimately tied to each other. If the private key is used to encrypt a piece of data, only the associated key can decrypt it. Conversely, if the public key is used to encrypt, only the private key can decrypt. The former allows a piece of data to be definitively tied to the owner of the public key that decrypts it. If the decryption fails, then the message was not encrypted by the owner associated with the public key. In the latter situation, when the public key is used to encrypt, then only the owner of the associated private key may decrypt, so one and only one person can read the data.

For our protocol, every peer could distribute their public key to each neighboring peer. The RSA cryptosystem could then be used for signing and verifying the message data. The sending peer would hash the original message, then sign the hash using their private key, and the RSA encryption method, and attach the computed hash as a HTTP header in the request. Each peer who receives the request simply has to match the sender's IP to their public key, and use it to verify the signature. This signature information could be logged to a file, along with IP, time stamp, port, and URL requested and used in all subsequent HTTP headers, so that the originators identity is preserved. The pycrypto library provides all the necessary functions to accomplish this.

### 5.2.3 Peer selection by a variety of metrics

So far we have discussed a few ways to evaluate and choose between peers. The decision is not one to one, a coordinator must asses a given peer by a variety of metrics. These include estimated bandwidth, connection strength/reliability, and trust factor. But there exists a balance between speed and security. Certainly, a peer that

[3]As no such case of copyright infringement by one party using multiple routers has occurred, there is no precedent to cite, so the outcome of such a trail is difficult to predict.

is well trusted and has a high estimated bandwidth would be a good candidate for an aggregation session, the two factors might not always balance. The coordinating peer must decide between a fast peer who is untrusted, and a slow peer who is well trusted. Although the goal of bandwidth aggregation is speed, we cannot ignore the inherent need for a secure system. So in the aforementioned scenario, an untrusted peer would not be picked.

Reliability is another factor that will influence peer selection. Peers will be monitored for timeouts. If a peer fails to response to a resource request within a dynamically computed threshold (expressed as a worst case download time), the peer will be dropped for the session. After the session is ended, the number of timeouts the coordinator experienced with that peer will be recorded. The transaction record for each peer in a given session will look like this: peer IP.

# 6 Conclusion, Future work

# Todo list