A Transport Layer Abstraction for Peer-to-Peer Networks

Ronaldo A. Ferreira, Christian Grothoff and Paul Ruth
Department of Computer Sciences
Purdue University
{rf,grothoff,ruth}@cs.purdue.edu
http://www.gnu.org/software/GNUnet/

***Abstract*— The initially unrestricted host-to-host communication model provided by the Internet Protocol has deteriorated due to political and technical changes caused by Internet growth. While this is not a problem for most client-server applications, peer-to-peer networks frequently struggle with peers that are only partially reachable. We describe how a peer-to-peer framework can hide diversity and obstacles in the underlying Internet and provide peer-to-peer applications with abstractions that hide transport specific details. We present the details of an implementation of a transport service based on SMTP. Small-scale benchmarks are used to compare transport services over UDP, TCP, and SMTP.**

I. Introduction

Peer-to-peer networks are typically overlay networks that are built on top of the existing Internet infrastructure. In an ideal overlay network, every node can communicate with every other node. However, this is not always the case with the modern Internet. Firewalls, network-address translation (NAT) devices, and dynamic IP assignment via DHCP are create obstacles that global peer-to-peer applications need to overcome. One central design goal for a peer-to-peer framework must thus be to virtualize the network and give the application a view of a uniform address space and communication model. While it may not always be possible to guarantee connectivity from every node to every other node, the details about the implementation of the transport layer should clearly be hidden from the application.

Another important problem with communications over the Internet is the increasing interference of governments in the activities of their citizens. China, for example, blocks access to some news sites hosted outside the country [3]. It is desirable for a peer-to-peer system to offer transport protocols that can be used in spite of these circumstances. UDP and TCP can easily be blocked based on the port number associated with a specific application; on the other hand, some protocols, such as SMTP, cannot be conveniently blocked without interfering with a significant portion of users.

It should be clear from the discussion above that one of the most important design requirements for a peer-to-peer system is the support for a wide variety of transport mechanisms. The goal is that a transport abstraction should support the full spectrum of transport services. These services may be unidirectional or bidirectional, stream-oriented or record-oriented, reliable or unreliable, and low-latency or high-latency. In particular, it is desirable to support a mixture of these different features in the same peer-to-peer network. In fact, two peers A and B may want to use different modes of communication on the same link. For example, suppose node B is behind a NAT box and cannot be reached directly via UDP or TCP. In a system with multiple transport protocols, A could initiate a connection by sending an e-mail to B (SMTP) and then have B contact A via TCP, allowing A to continue further communication on a bidirectional TCP connection.

We will use GNUnet as our reference peer-to-peer system, but it should be clear that the idea of a transport abstraction can be applied to other systems. GNUnet is a peer-to-peer framework whose main focus is on security [1], [5]. The goal of the GNUnet project is to become an important tool for free information exchange in a world which is hostile toward uncontrolled communication. GNUnet's primary design goals are to protect the privacy of its users and to guard itself against attacks or abuse. GNUnet does not have any mechanism to control, track or censor users. Instead, the GNUnet protocols aim to make it as hard as possible to identify specific network activities or to disrupt operations.

In this paper, we present a transport layer abstraction for GNUnet and benchmarking results that evaluate the performance of the corresponding UDP, TCP and SMTP transport implementations. We will describe the SMTP transport implementation in more detail, since this is the less conventional choice.

II. GNUnet and The Transport Layer

GNUnet is a peer-to-peer framework that can support multiple applications [5]. It has a layered design (see Figure 1), and normally runs as a daemon. The applications talk to the GNUnet daemon (`gnunetd`) in a client-server manner using TCP connections. Current examples of applications are anonymous file sharing (AFS) and a message exchange program (Chat). In GNUnet, the server `gnunetd` is responsible for accounting, routing and link encryption. The core relies on implementations of the `TransportAPI` for the actual transport of messages.

In GNUnet, peers are identified by *node identities*, which are the hash codes of their public keys. The GNUnet core provides link-to-link encrypted transmission of messages to other nodes that are identified by this node identity. The core binds each node identity to an address in the underlying Internet. The nature of these network addresses depends upon which transport mechanism that is used. For example, for UDP and TCP, IP address and port number are used. For SMTP, an e-mail address is used. The core is responsible for link-to-link encryption, binding of node identities to routable addresses, and peer discovery. This leaves the peer-to-peer applications concerned only with node identities. Due to space limitations, we refer the reader to [5] for more details about GNUnet and its various protocols; for the remainder of this paper, we will concentrate on the transport abstraction.

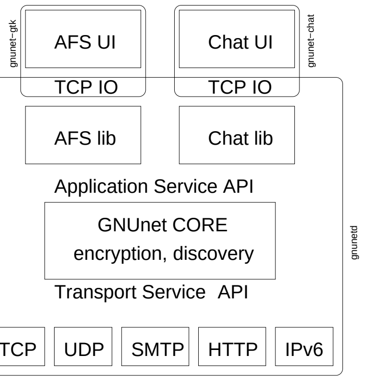

Fig. 1. GNUnet layers.

Node to node communication in peer-to-peer networks is inherently unreliable. In contrast to client-server architectures, node failure is part of the normal mode of operation. But even if nodes do not fail, the transport layer may be built on top of an unreliable communication protocol such as IP or UDP. The design question in this case is whether or not the transport layer implementation should hide this fact and guarantee delivery if the other node is reachable. In other words, the question is whether or not the transport layer or core should provide reliable communication like TCP and hide the unreliability of the network, or if all network problems should be exposed to the application.

In peer-to-peer systems, it is better to expose the unreliability of the transport layer to applications. There are multiple reasons for this. For instance, it forces the programmer to assume that every communication can fail. This is a good idea since a reliable transport abstraction may give application designers a false sense of reliability that cannot fully be achieved since peers can still fail. Another reason is that many applications may not require reliable communications; for example, a flooding search may send out 12 queries in parallel, and if one of them is lost on the transport layer, it is still possible that the remaining 11 queries will return a sufficient number of results. Adding retransmission on the transport layer in these cases merely increases the overhead without providing any major benefit.

The same rationale applies to the question of ordered delivery. Choosing the weaker semantics (no guarantee for order of delivery) makes the transport layer cheaper and more resilient to attacks. For both reliability and message ordering, the transport layer implementation may still use an underlying protocol such as TCP that has stronger semantics; This might happen, for example, because the network or the host configuration does not allow the use of cheaper protocols such as IP or UDP).

```
typedef struct {
    TSession        * tsession;
    HostIdentity      sender;
    p2p_HEADER      * msg;
    unsigned int      size;
    int               isEncrypted;
    int               crc;
} MessagePack;
typedef struct {
    unsigned int      version;
    HostIdentity    * myIdentity;
    void (* receive)(MessagePack * mp);
} CoreAPIForTransport;
```

Fig. 2. Data structure used in the interface core/transport layer.

In order to allow using any common transport protocol as the basis for the transport layer, the size of the messages exchanged between the core and the transport layer must be either fixed (and in this case equal to the smallest value among the transport MTUs) or communicated from the transport layer to the core. While UDP can technically support datagrams of up to 64KB, the operating system may impose a smaller MTU (see RFC 1122 [2], for example). Fragmentation considerations often dictate even smaller message sizes, such as 1472 bytes on ethernet. Other protocols, like SMTP, have no restrictions on the message size but may have a high per-message overhead. Therefore, we advocate the idea that the transport layer should advertise a transport-specific MTU to the core. The core is then responsible for queuing multiple application messages into a larger message (which must be smaller than the MTU) and using the transport layer to transmit this message. The smallest MTU should ideally be larger than the largest single application message, but if this is impossible, the core could provide fragmentation and reassembly.

A. *The core API*

The GNUnet core provides a very simple interface, the `CoreAPIForTransport` data structure, to the transport layer. It consists of the method `receive` which is invoked by the transport layer whenever a message is received. The core API data structure also contains a version number and the identity of the local node. Figure 2 shows the data structure. A pointer to this data structure is passed to an initialization method (`inittransport`) that the transport layer implementation must implement. The `inittransport` method then returns a data structure with the methods that the core can invoke on the transport layer (see section II-B).

The `receive` method from the core interface takes one message as its argument. The message contains a transport session handle (*TSession*). This handle is used to identify a transport session (e.g. a TCP stream). The *TSession* contains the type id of the transport and an opaque handle that the transport layer is free to define. The transport layer can pass NULL for the session handle if the specific transport implementation has no notion of sessions.

The second field (`sender`) is the node identity of the sender of the message. The transport layer must communicate this information, but the means by which this is communicated are up to the transport implementation. For example, UDP messages

```
typedef struct {
    unsigned short protocolNumber;
    unsigned short mtu;
    unsigned int cost;
    int (*verifyHelo)(HELO_Message *helo);
    int (*createHELO)(HELO_Message **helo);
    int (*connect)(HELO_Message *helo, TSession **tsession);
    int (*send)(TSession *tsession, void *msg,
                unsigned int size, int isEncrypted,
                int crc);
    int (*associate)(TSession *tsession);
    int (*disconnect)(TSession *tsession);
    int (*startTransportServer)(void);
    int (*stopTransportServer)(void);
    void (*reloadConfiguration)(void);
} TransportAPI;
```

Fig. 3. The transport API.

contain the sender identity in every message, whereas in the TCP implementation, the client sends its identity only when the connection is established (but never thereafter).

The remaining four arguments describe the message itself. They are all passed to the transport layer by the core. They consist of a pointer to the message itself, the size of the message, a flag specifying whether the message is in plain text or encrypted, and the CRC32 checksum of the plain text message. The transport layer is not concerned with encryption or with verification of the checksum; it is only concerned with transmitting these four pieces of information. The details about how this information is communicated are again transport-specific and not specified.

B. The Transport Interface

The `TransportAPI` is again a data structure containing a list of function pointers, see Figure 3. It provides access to three fields and nine methods which extend GNUnet with a new transport service. A new transport mechanism can be integrated into GNUnet by building a dynamic library `libgnunettransport_XXX` that exports a function `inittransport_XXX`. If GNUnet finds a command to load the transport named `XXX` in the configuration file, it will load this dynamic library and invoke the `inittransport_XXX` method passing the core API described in the previous section. `inittransport_XXX` is then supposed to return a pointer to the `TransportAPI` data structure, or NULL on error.

The transport API data structure contains three fields. The first field is the protocol number, which is a unique, small number that can be used to identify the transport protocol inside of GNUnet. The second field is the maximum transfer unit (MTU) supported by the transport implementation. If a GNUnet transport implementation is stream-oriented (like TCP) and has no obvious limit on the message size, the MTU should be chosen such that a reasonable trade-off between per-message overhead, IP-level fragmentation and latency is achieved. The third field defines the *cost* of using this transport mechanism compared to other services. For example, UDP is typically cheaper than TCP, and TCP is much cheaper than SMTP. Given the choice, the GNUnet core will choose the cheapest available transport mechanism to send a message.

The first function that GNUnet will typically call on the transport API is `startTransportServer`. This function is called when GNUnet is ready to receive messages from the transport layer. It should be used to start a server thread that, for example, listens on some port for incoming messages. When GNUnet is stopped, it calls `stopTransportServer` in order to give the transport a chance to shutdown properly.

GNUnet uses so-called *HELO* messages to notify other nodes about available transports. A *HELO* message contains, among other things, the transport protocol number and the address of the transport endpoint. Since the GNUnet core does not know anything about the addressing scheme used by the transport mechanism (it could be anything, from an IP address and port number to an e-mail address or an IPv6 address), it calls `createHELO` on the transport mechanism in order to obtain this information. *HELO* messages are of variable size with the transport-layer address being the variable-length field `senderAddress` at the end of the message. `createHELO` is supposed to fill in the fields `senderAddressSize`, `protocol`, `MTU` and `senderAddress`. `createHELO` returns `OK` on success. `createHELO` can fail; for example, it may fail if the transport service is unidirectional (send only) and cannot be advertised as an address under which the local node can be reached. This might, be the case, for example, with a TCP service behind a NAT box. If the TCP implementation is aware of the NAT problem, it may decide to always return `SYSERR` in `createHELO` and thus ensure that other nodes will never attempt to initiate a TCP connection. In this case, the TCP service would only be used for outbound connections by GNUnet.

The function `verifyHelo` is a counterpart to `createHELO`. It is invoked whenever GNUnet receives a *HELO* message with the same transport protocol id. `verifyHelo` is supposed to check whether the sender address specified in the *HELO* conforms to the standards set for sender addresses in the specific transport mechanism. `verifyHelo` does not have to verify if the rest of the *HELO* is well-formed or if the address is actually reachable. `verifyHelo` may check if the user has configured the transport mechanism to specifically deny sending messages to that address (a blacklist mechanism: this may, for example, prevent attempts to connect to a 10.0.0.0 IP network).

The remaining four methods are the ones actually concerned with sending messages. `connect` is used to establish a connection to a remote node. `connect` is passed a *HELO* message and, if successful, is supposed to set its second argument to the transport session (`TSession`) that was established. The `TSession` handle is used by the core in calls to `send` to identify where to send the message. The other arguments to `send` are the message itself, the size of the message, the indication of whether the information is encrypted, and the checksum of the message in plain text. These provide the transport mechanism with all four arguments that must be passed to the `receive` function of the core at the receiver node. The `disconnect` method does the opposite of `connect`; it shuts down a connection and deallocates the resources associated with it. In particular, the transport implementation must free the `TSession` data structures that were created in `connect`.

While it seems that the set of functions described so far would be sufficient, we need one additional method, `associate`, in order to handle bidirectional transport mechanisms. The problem with bidirectional transports is that a session is not always initiated by the local GNUnet core calling `connect`, but may instead have been started by a remote node. The GNUnet core first sees the session when the transport layer calls `receive`. At this point, two things may happen. Either the core decides to use the bidirectional session to send replies, or the core uses another (potentially cheaper) transport mechanism to answer, if it answers at all. If the core wants to use the existing session to send replies, it must retain the transport session handle. Without `associate`, this would cause problems because the transport layer would not know when to close the connection and free the resources. Thus, we require the core to invoke `associate` if it decides to use the transport session for further communication. `associate` is similar to `connect` in that it ensures that the session handle is valid until the core calls `disconnect`. The implementation of `associate` is similar to reference counting in garbage collection. `connect` and `associate` increment the reference counter; `disconnect` decrements it.

Another important detail in the implementation of bidirectional transports is that they must time out stale connections. The current interface would allow nodes to establish a connection that would last indefinitely without being used. For example, the core of node A may decide to establish a cheaper way to communicate and not to `associate` with an inbound connection. The transport of node A cannot instantly close the session since the other node B may still be using it to send data. But B may keep the connection open for the same reason; it may expect to receive data from A on that connection. Thus, the transport service must time out sessions that are not associated with the core for writing.

C. Security Considerations

An interesting security problem in peer-to-peer networks arises when malicious nodes advertise invalid or incorrect peer addresses. For example, it would be possible in Gnutella [6] to advertise `example.com` as a peer; even the port can be freely chosen in the advertisement. If peers spread this advertisement and frequently attempt to connect to this host, the peer-to-peer network could become a tool for a distributed denial of service attack. On the other hand, without a central server, the ability of peers to advertise other peers cannot be avoided.

Our solution to this problem is that every peer A that receives an advertisement for another peer B must check that the advertised address is valid by sending a PING message containing a *challenge* (a randomly chosen integer) to the advertised peer B. If B receives the PING, it responds with a PONG message which also contains the challenge, confirming that it can be reached under this address. Only after this protocol has been run should A notify other peers of B's existence. This prevents a malicious node M from advertising a non-participating third party T on the network since T would not properly respond to A's PING. M also cannot easily fabricate a PONG for T because the message sent to B contains a challenge which is unknown to M. While M has tricked A into sending a single message (the PING) to T, this cannot be used to seriously attack T since M had to send a message to A first. If M had sent the message directly to T, it would have caused an equal amount of traffic. The only gain that M has potentially achieved is that it was able to hide its identity from T.

III. Example: SMTP Implementation

When GNUnet starts running, it loads all the transport modules defined in its configuration file. During this process, the initialization code of the SMTP transport opens a connection to an SMTP server (sendmail, qmail, etc.) that is running either on the local host or remotely; this connection will be used to send messages to the other peers. Observe that GNUnet does not establish a direct SMTP connection to the other peers, but relies instead on existing mail transfer agents (MTAs) to send the messages.

A. Sending E-mail

When the SMTP transport service receives a message from the GNUnet core, the message is extended with a header that contains the node identity of the sender and the meta-information provided in the parameters of `send`. The resulting message is base64 encoded, encapsulated according to the MIME conventions [4], and sent to the MTA over the pre-existing TCP connection. Most MTAs store the mail on the drive before sending an acknowledgement to the client in order to ensure guaranteed delivery even after a crash. While this is not required for GNUnet mail (especially since the semantics only specify unreliable communications), this disk I/O is a significant overhead for the SMTP transport. The MTA then resolves the destination address using DNS (MX record) and contacts the remote mail server, which again receives the message via SMTP and initiates delivery.

B. Receiving E-mail

In order for GNUnet to receive an inbound e-mail, the mail must first be delivered to the local machine. If the local machine is the receiving host according to the MX record for the e-mail address, this step is handled by the SMTP protocol. But in the case where the GNUnet node runs behind a NAT box, the mail will typically be stored on the mail server at the ISP. In this case, the host will periodically poll for new mail, for example using a POP client. Under this last configuration, the polling interval will be a major contributor to the delay in the SMTP transport. For GNUnet to work properly, we assume that one minute is a reasonable interval. Polling with POP can easily be automated using `fetchmail`, a tool that is available for most UNIX systems.

In many cases, this is not the only problem. Normally users will have only one e-mail account available. Thus it is necessary to filter the inbound GNUnet messages from the other messages that are destined for the user. Since we do not want to tag all GNUnet e-mails with a uniform header (this would make it too easy for adversaries to filter and effectively censor GNUnet traffic), the advertisement for the SMTP address of the peer does not only contain an e-mail address but also a *filter* line. The sender is required to add this line to the header. Since the receiver of the e-mail specified which filter to use, `procmail` can be used to distinguish mails that have the appropriate filter line. The user can change the filter line whenever he

```
:0:
* \^X-mailer: GNUnet
/tmp/gnunet.smtp
:0:
/var/spool/mail/$USER
```

Fig. 4. Example procmail configuration with “X-mailer: GNUnet” as the filter line and “/tmp/gnunet.smtp” as the name of the pipe.

wants; it will, of course, take some time to propagate the new address information into the network.

Finally, `procmail` needs to be informed of how to deliver the GNUnet mail to the `gnunetd` process. The easiest way is to use a named pipe (fifo). The user specifies in the GNUnet configuration the name of the pipe, and `procmail` writes the filtered mail into that pipe. The SMTP transport then reads the mail, decodes the base64 encoded body and forwards the message to the GNUnet core. An example `.procmailrc` configuration file is given in Figure 4.

C. Security considerations for SMTP

The primary security problem with SMTP is the potential for harassment of users. Other transports (UDP, TCP, HTTP) have this problem to a much lesser extent. While sending massive amounts of traffic can become an attack with every transport protocol, fairly moderate amounts of data can become a problem when sent to a user via e-mail, especially if the user is not educated enough to filter the spam. Still, it is possible to use SMTP as one possible transport mechanism for peer-to-peer networking. Since GNUnet is completely decentralized, a solution to the security challenge requires that peers be able to advertise e-mail addresses of other peers on the network.

In order to prevent attacks, every peer first validates the advertisements before using the e-mail address for actual transmission or advertising it further. The validation procedure requires the peer to send an e-mail message containing a GNUnet PING message together with advertisements for its own transports to the peer. If the peer responds with a proper GNUnet PONG message, the address is considered valid. Every PING message contains a random number (challenge) that the responding peer must copy into the PONG message. This makes it impossible to send a fake PONG reply for anybody that does not control a router on the path between the two peers. Note that the PONG message does not have to use the SMTP transport mechanism; any known transport for the peer will do.

This mechanism ensures that a malicious peer that sends an advertisement for an invalid (non GNUnet) e-mail address will trick the receiving peer into sending at most one small message to that address. The bandwidth that the adversary spends on sending advertisements is thus proportional to the amount of e-mail that the victim receives. More importantly, the adversary is not anonymous. While the victim does not receive the mail directly from the attacker, it is clear that the attacker is the node sending the advertisements since no honest node will send advertisements without having received the PONG confirmations. Thus, it is possible to track down the attacker.

Transport	UDP	TCP	Purdue	RH 8.0	qmail
11 bytes	31 ms	55 ms	781 s	89 s	24 s
407 bytes	37 ms	62 ms	789 s	98 s	25 s
1,221 bytes	46 ms	73 ms	804 s	98 s	25 s

Fig. 5. Time to send 1000 messages (transport only).

A more sophisticated attack involves mailing lists. The problem here is, that an adversary could subscribe to a mailing list and then advertise the address of the mailing list on GNUnet. Peers would send mail to the list and the adversary could send responses to the PING messages since he is one of the recipients. Since the peers can confirm that the address is valid, they would now start advertising the address, causing even more traffic for the list. In this way, an adversary could anonymously drown any open mailing list in unsolicited traffic. The solution to this problem is to ensure that GNUnet SMTP traffic will not be forwarded by any modern mailing list software. This can be achieved by making every GNUnet e-mail look like a bounce message [7]. Bounce messages are used to notify the sender of an e-mail about an invalid or unavailable receiver address. Since mailing lists often have the problem that one of its members is unavailable, it is safe to assume that bounces are always filtered.

IV. Performance Measurements

For the measurements, GNUnet was configured to only use one transport service for the benchmarks; automatically switching to a more efficient protocol was thus impossible. In a first benchmark, the latency of the UDP, TCP and SMTP transport services was measured. For this, the service was called to send a message to itself. This gives a network-independent approximation of the overhead of the service on the peer itself. The loopback measurements of the SMTP transport were performed on three different machines spanning a range of modern SMTP configurations. We used a Pentium III 800MHz running RedHat 7.3 with the *Purdue* Computer Science configuration which includes filters for spam. We also used a Xeon 2GHz with a vanilla *RedHat* 8.0 sendmail configuration. Furthermore, we used qmail on a Pentium III 1GHz running Sorcerer GNU Linux (*SGL*). The numbers for UDP and TCP are provided using the SGL configuration. The qmail benchmark uses qmail's internal filtering whereas the sendmail benchmarks use `procmail` to filter and deliver the mail. We used the transport layer to send a message of b bytes (excluding transport protocol headers) directly to the local machine. This way, network latency and packet losses on the wire have no impact on the time measurements. n messages were sent sequentially over the transport layer, sending message $i+1$ after the i-th message was received. All messages were sent over the same connection and the time to establish the connection was not taken into account since this overhead is minuscule in practice – as long as a connection is used for a significant number of messages. The benchmarks show that UDP and TCP are, as expected, both significantly faster compared with any of the SMTP services (in particular with the qmail benchmark that was run on the same machine and is recognized to be much faster than sendmail). Among the SMTP implementations, there can be significant differences depending on the SMTP configuration. Filtering with an external tool like `procmail` that needs to reload and parse its configuration for each mail can be very expensive. Applying spam filters can also significantly impact the performance of the underlying SMTP implementation. The micro benchmark shows that SMTP can be a viable solution for initiating peer-to-peer sessions: a few seconds to connect to a peer will probably not even be noticed by users.

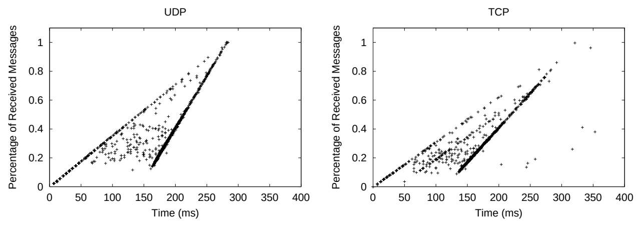

Fig. 6. Sending 1,000 messages of 1,200 octets payload with the unreliable UDP and TCP transports without spacing.

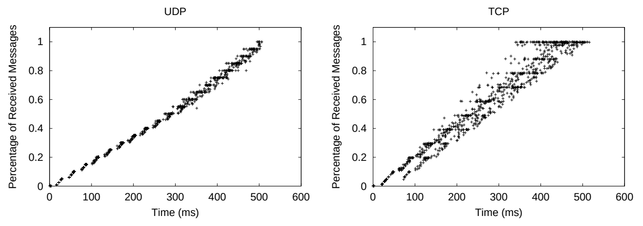

Fig. 7. Sending 1,000 messages of 1,200 octets payload with hand-optimized spacing between message-trains.

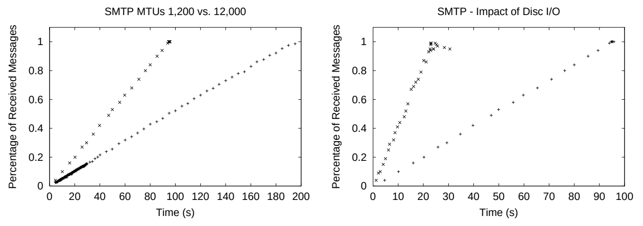

Fig. 8. SMTP throughput (with hand-optimized spacing to avoid thrashing) for the same total payload with different MTUs and with/without RAM-disk for the mail spool.

The second set of benchmarks measures the possible throughput for a transport. Throughput can be measured by sending multiple messages in parallel and measuring packet loss. Note that not only UDP but also the TCP transport can actually lose messages, since the TCP implementation drops messages if the `write` to the socket blocks. For this benchmark, we report the message loss after allowing t seconds for sending m messages. If messages were not sent (or received) after t seconds, they were considered lost. The benchmark was performed using two Xeon 2 GHz machines running RedHat 8.0 with sendmail. The machines were connected via a direct 100 MBit ethernet connection. Figure 6 shows the percentage of 1,000 messages with a payload of 1,200 octets that could be received after t seconds if the messages were passed to the kernel without any artificial delays. The maximum throughput in this case is about 4,169 kilobytes per second (kbps) for UDP. Since the application does not limit the transmission rate, the Linux kernel starts dropping messages shortly after the start of the benchmark. This allows the application to finish sending all the messages earlier – the processing time of a dropped packet is smaller – but the loss rate can become quite dramatic, often exceeding 70%. For TCP, the situation is the same, only that due to the inherent protocol overhead, the throughput is slightly worse with at most 3,627 kbps. The high packet loss that occurs in this naive design can be avoided by reducing the rate at which the application sends messages. Figure 7 shows the results that can be achieved if the application sends the packets in trains with hand-tuned spacing between the trains. Conceptually, the delays are equivalent to congestion control in TCP, but since the timers available to the application are more coarse grained, the performance is worse than in real TCP, even if hand-tuned for a static testbed. Note that TCP is a bit easier to tune since the TCP windows with the better timers available to the OS can help leverage the coarse-grained application level timers. Thus, with up to 3,310 kbps throughput, TCP throughput is (while more random) a bit better than the 2,343 kbps of UDP.

For SMTP, we only show the rate-controlled numbers. Figure 8 shows that the SMTP throughput for messages of size 1,200 octets is 6 kbps. The high per-message overhead can be alleviated by increasing the message size to 12,000 octets, resulting in 13 kbps throughput. A major bottleneck in this case is sendmail writing every message onto the drive when the message is queued. By using a RAM-drive for the mail queue (violating SMTP reliability in the case of a machine crash), the throughput can be increased to 51 kbps, which is still about 50 times slower than UDP with spacing. As shown by the loopback-benchmark, other MTAs may have different performance characteristics.

V. Conclusion

We have presented the design of a transport abstraction for peer-to-peer systems. The abstraction can support a wide range of underlying transport mechanisms and we have implemented service modules for UDP, TCP and SMTP. While the benchmarks clearly show that SMTP is significantly worse in terms of performance, the service can still be useful to initiate connections and negotiate the use of a cheaper service. We have addressed security concerns that arise with the use of SMTP and argued why a peer-to-peer transport abstraction should have unreliable semantics.

References

[1] K. Bennett, C. Grothoff, T. Horozov, and I. Patrascu. Efficient Sharing of Encrypted Data. In *Proceedings of ASCIP 2002*, 2002.
[2] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, IETF, October 1989.
[3] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, San Francisco, CA, 2002.
[4] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045, IETF, November 1996.
[5] The GNUnet Project. http://www.gnu.org/software/gnunet/.
[6] Gnutella. http://gnutella.wego.com.
[7] K. Moore and G. Vaudreuil. An Extensible Message Format for Delivery Status Notifications. RFC 1894, IETF, January 1996.