Short Messages

By Damon Wischik†
UCL

This paper has three purposes. The first is to explain to a general audience what is involved in retrieving a web page or performing some other complex network task, and what can make it slow, and why the problem of slowness is likely to get worse as networked applications become more complex. The second is to describe to those who program networked applications certain facts that we have learnt from modelling communication networks, which may allow more efficient applications to be written. The third is to describe to network modellers an interesting class of problems relating to algorithm design for communication networks.

1. What causes delay in network transactions?

The time it takes for a light pulse to travel from London to New York and back again is around 38ms. With a perfect network, the time it should take to request a web page from `nytimes.com` and to receive the reply should be 38ms, plus a handful of milliseconds for the web browser to formulate the request and the webserver to formulate the reply. In practice, depending on the time of day, it can take up to five *seconds*. What causes this delay? Cohen and Kaplan [2] give a full account; here are the highlights.

Anatomy of a website. When a web browser sends a request for, say, `http://gmail.com`, the server does not typically reply with the entire page. Instead it returns a shell page with some plain text, plus links to further items such as pictures or style information. The web browser reads the shell page, works out what further items are needed, and automatically sends out more requests. These new items may in turn request further items. Many modern web pages link to code which is retrieved and executed on your web browser, which then issues further requests—this feature is used by interactive web sites, where new content is retrieved when the user types or clicks a button. Sometimes the server doesn't even serve any content at all: it may simply reply saying "the item you requested has moved and is now at x", and the web browser automatically sends its request to the new location. The status bar at the bottom of a web browser window shows which items it is busy retrieving.

Figure 1 shows the graph of dependencies in the web page that was retrieved when I logged on to `gmail.com` on 28 August 2006. (The labels in this figure are simple mnemonics.) The very first item requested is `ServiceLoginAuth`, and the web server replies with a redirection to `CheckCookie`, which then invokes the retrieval of `auth`, which invokes the retrieval of `browser.js`, and so on. The dependency graph is not a strict tree: the code says to retrieve `loading html` and `hist1`, and only when both of these have been retrieved will it go on to retrieve `hist2`.

Some of the items in this dependency graph can be retrieved in parallel, such as the 42 images (status flags, rounded corners, unread mail icons, etc.), whereas others must be

† Damon Wischik is supported by a Royal Society university research fellowship

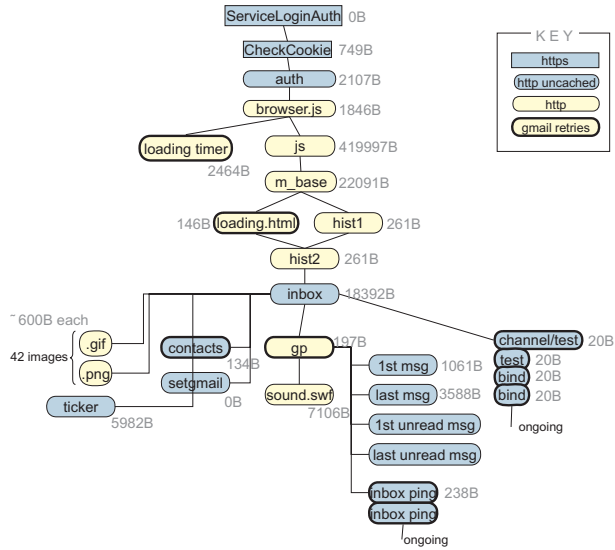

Figure 1: The graph of dependencies for retrieving an inbox at `gmail.com`, gathered on 28 August 2006 using Firefox 1.07. Paros 3.2.13 was used to trap each http request. On successive visits, different sets of requests were let through; the set of ensuing requests reveals the graph. The numbers show the size in bytes of each item.

retrieved in sequence. Even when items can be retrieved in parallel, the web browser may choose not to do so—see the table on page 11. All this back-and-forth leads to delay.

The web browser can choose to cache some of the items, so that on future visits to `gmail.com` they do not need to be retrieved over the Internet. The web server indicates which items it is safe to cache and for how long.

The http and https protocols. Now we drill down into the mechanics of how the web browser actually retrieves an item, and find more opportunities for delay. The web browser and server communicate by sending messages to each other. To retrieve a single item, there is a round of messages:

	Client	*Server*
1.	Hello!	
2.		*Hello!*
3.	Please send X.	
4.		*Here's X. Goodbye!*
5.	Goodbye!	
6.		*OK!*

To retrieve an item over a secured connection (`https`), as for the first two items in Figure 1, step 3 is replaced by some extra steps which in the best case are

	Client	*Server*
3a.	Let's use cipher c.	
3b.		*Here's my public key.*
3c.	Here's a secret key for all further data.	
3d.		*Ready.*

3e. Please send X.

Reliable delivery over an unreliable network. Each one of these messages is broken down into packets† and send over the Internet. The Internet is inherently unreliable and may drop packets en route. To cope with this, the computer's operating system runs the Transmission Control Protocol (TCP) which implements reliable delivery as follows.

TCP sends the first packet of a message, and waits for an acknowledgement that it has been received (except for the final OK! which does not expect an acknowledgement). It sets a retransmission timeout RTO, and if no acknowledgement is forthcoming within RTO then it retransmits. When it receives an acknowledgement it goes on to transmit the remaining packets, usually sending several packets at a time, and it has a more rapid means of detecting when it needs to retransmit. For the final packet in a message, the acknowledgement may be wrapped up with the first packet of the reply message.

Ideally RTO might be roughly the round trip time from one party to the other and back again, but round trip time varies from packet to packet so RTO is continually updated to take account of estimated mean and variability. Each side of the communication maintains its own copy of RTO. The complete specifications of how RTO is updated are in [11], and a survey of how it is implemented in practice is given by Rewaskar et al. [12]. Broadly speaking, if the round trip time experienced by a packet is RTT, then

(i) RTO=3 seconds for the first packet in a round of messages
(ii) RTO=$\max(3\mathsf{RTT}, 200\text{ms})$ for the second packet
(iii) RTO decreases to $\mathbb{E}\mathsf{RTT} + \max(10\text{ms}, 4\mathbb{E}|\mathsf{RTT} - \mathbb{E}\mathsf{RTT}|)$ thereafter, except it is not allowed below 200ms (the specification says the minimum should be 1 second)
(iv) RTO is used as the timeout the first time a given packet is retransmitted, 2RTO the second time, 4RTO the next, and so on.

There are yet more layers under TCP in which delay can be introduced. Suppose the web-browsing computer is connected to the Internet over wifi: then its wireless card and the wifi base-station will run yet another protocol with timeouts and retransmissions. The net effect is that TCP is shielded from most dropped packets, but at the expense of increasing its estimate of round trip time mean and variance—which means it takes longer to recover when it actually does suffer a dropped packet.

Reliability can also be implemented in higher layers too. In Figure 1, there are several items with bold borders—this denotes the fact that Google's code repeatedly attempts to retrieve the item if it has not received within a few seconds. *Network engineers normally think of reliable delivery as a network function, but Google has implemented it in Javascript, presumably because the network does not do what Google needs.*

Sequential versus parallel programming. Here is a final example of a distributed system: obtaining a directory listing in Windows, over a virtual private network.

	Client	*Server*
1.	Hello!	
2.		*Hello!*
3.	What is first file in directory d?	
4.		*It is* f_1.

† Packets are typically of size $\leq$ 1500bytes for ethernet, $\leq$ 576bytes for a modem. Messages 1, 2, 5 and 6 always fit in a single packet.

5.	What is the next file after f_1?	
6.		*It is f_2.*
	⋮	
$n+4$.	What is the next file after f_n?	
$n+5$.		*No more files.*
$n+6$.	Goodbye!	
$n+7$.		*Goodbye!*
$n+8$.	OK!	

Each transmission apart from step $n+8$ uses an RTO. The task is inherently parallel, but the Windows programming interface turns it into a sequential computer program. This is a natural way to program, but it leads to systems which turn sluggish when the network is slow or has too many dropped packets.

Growth in complexity. It seems likely that network tasks will become more complex: web pages will become richer and more interactive, and web browsers will become tools to mash up data from a whole host of networked databases. The more complex the task, the more sensitive it will be to latency and packet drops.

2. Internet traffic statistics, and how to make TCP faster

If the first packet in a message is dropped, then the TCP's RTO timeout is at least 200ms, and can be as much as 3 seconds. If the message is more than one packet long then TCP waits for the other party to acknowledge the first packet before transmitting the rest.

It would be easy to speed up TCP. We could send multiple redundant copies of each packet back-to-back, so that even if one packet is dropped there's a good chance another copy will make it through. We could send later packets in a message at the same time as the first, without waiting for an acknowledgement. These changes to TCP would make it more aggressive, and more likely to cause congestion. In fact the original design of TCP was more aggressive, and in 1988 this led to Internet congestion collapse [6]. That experience resulted in the current design of TCP, and a strong disinclination to experiment with making TCP more aggressive.

I will argue that the characteristics of Internet traffic mean that it is safe to send several redundant copies of packets that make up short messages, say messages of one to three packets, though of course there is no point sending them so close together that they share the same fate. Note that many of the items in the `gmail.com` inbox are very small, so they would benefit.

There has also been a proposal that TCP should be allowed to send up to four packets or so without waiting for an acknowledgement [1]. This option is widely implemented, but it seems to be turned off by default.

(*a*) *Heavy tails and short messages*

Most traffic is composed of large traffic flows, known as elephants. Elephants are large and rare. Yet most network tasks, are composed of short flows, known as mice. Mice are small and numerous. Since most traffic comes from elephants, capacity planning is based on elephants, but it is the mice that users value most highly. This is the central paradox of communication network design.

What do users value? The Three mobile phone network pay-as-you-go charges in October 2007 are £1 per megabyte for broadband data, 50p per minute for video calls of about 64 kbit/s (£1.04 per megabyte), and 12p per text message of 140 bytes (£857 per megabyte). And if a picture is worth a thousand words, of average length 5 characters, then there is no point sending JPEG files any larger than 6 kB, much less the ≈4 MB photos taken by a digital camera.

Why the dichotomy between mice and elephants? Measurements of Internet traffic have found that message sizes have a heavy-tailed distribution [3]. This is a class of probability distributions for which a few large items are likely to outweigh many many small items. These distributions have been found in biology, chemistry, ecology, finance, etc. Mitzenmacher [10] gives some of the history, and describes three general models of why they might come about; for the particular case of web file sizes, Doyle and Carlson [4] suggest that heavy tails arise as a consequence of optimal partitioning of data. In practical terms, the elephants today are from peer-to-peer sharing of video files, whereas the mice are short control messages and plain text.

Why do network engineers look at elephants? In the early 1990s, researchers at AT&T discovered that Internet traffic was self-similar (it has spikes at many timescales, 'peaks, riding on bursts, riding on swells') and long-range dependent (strong positive correlations) [8]. Researchers went looking for the cause, and found heavy-tailed message sizes, and proved that the aggregate of many heavy-tailed flows will lead to self-similarity and long-range dependence [15]. Self-similarity shows itself clearly in plots of network load, and long-range dependence has the consequence that networks need very large buffers.

How much traffic is made up of elephants? Table 1 is a dataset collected by Tanenbaum et al. [14], which lets us quantify just how big the elephants are compared to the mice, and which also shows that the elephants are growing. The data is of file sizes on a university filestore. Assume that this represents message sizes in the Internet† Now, consider making TCP more aggressive by sending three copies of the first packet of every message, two copies of the second, and one copy of subsequent packets—this crude hack would increase total network traffic volume by 34% in 1984, 2.7% in 2005. Or consider enlarging every flow so that it is at least five packets long—this would increase total network traffic volume by 21% in 1984, 1.3% in 2005. (There is however not yet any standard way for an application to tell the operating system whether a message will be short, other than handing over the entire message in one go.)

† I have used this data on file sizes, rather than Internet traffic statistics, because I have not found historical Internet traffic data which gives such fine detail about the distribution of small message sizes. Internet traffic engineers have been more interested in elephants than in mice.

size	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
vu1984	1.79	1.88	2.01	2.31	3.32	5.13	8.71	14.73	23.09	34.44
vu2005	1.38	1.53	1.65	1.8	2.15	3.15	4.98	8.03	13.29	20.62
size	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
vu1984	48.05	60.87	73.51	84.97	92.53	97.21	99.18	99.84	99.96	100
vu2005	30.91	46.09	59.13	69.96	78.92	85.87	90.84	93.73	96.12	97.73
size	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}		
vu1984	100	100	100	100	100	100	100	100		
vu2005	98.87	99.44	99.71	99.86	99.94	99.97	99.99	99.99		

Table 1: File size distributions, taken from Tanenbaum et al. [14]. The rows show the what percentage of files are smaller than or equal to the specified size in bytes. The data sets are from the Computer Science filestore at the Vrije Univeriteit, measured in 1984 and 2005.

Whereas network engineers concentrate on the elephants, for our purposes the mice are more interesting. In networks which are provisioned to carry elephants, it is safe to be aggressive when sending mice, e.g. by setting RTO smaller than the round trip time. The elephants get steadily bigger as years go by, and the mice can be more and more aggressive.

(*b*) *Packet drop statistics*

Packet drop rates vary wildly, by time and location, and congestion hotspots hop around the network. The snapshots at `http://www.internetpulse.net` show the recent packet drop rates at the interconnects between 12 major service providers, averaged over various time intervals; on 21 October 2007 at 01:30 GMT the median, mean and maximum across the various interconnects were

time period	median	mean	max
1 hour	0	0.20%	4.17%
4 hours	0	0.17%	3.12%
24 hours	0.03%	0.15%	2.60%

End-to-end packet drop rates will be higher. A 1999 measurement study [20] found packet drop rates from the US to Sweden of around 3%, and also measured the autocorrelation in packet drops, and found that it drops off quickly: conditional on a packet drop at time 0, the probability of a packet drop at time t is roughly

t	50ms	100ms	200ms	300ms	400ms	500ms
drop prob.	15%	7.9%	5.9%	4.9%	4.2%	3.7%

The wireless setting is completely different. A measurement campaign conducted in a Berlin machine shop in 2002 [18] found that cell drop probability was a few percent for much of the time, but there were 20 minute periods where it climbed to 60%, when a nearby machine was active. (This means incidentally that the conditional drop probability will be very high, a reflection of non-stationarity rather than correlation.) The wireless layer has its own retransmission protocol, which will shield most of this very high drop rate from TCP; TCP will see it instead as lowered capacity and more variable RTT.

Conclusion. The RTO retransmit timer can be set to 100ms, and probably less. There is probably no point setting it very much less than 20ms, since two packets that close together are fairly likely to share the same fate.

3. A model of computation for distributed algorithms

In Section 4 we will analyse three different network transactions. The first two are conceptually straightforward. The third is more elaborate; to understand where it's coming from it is worth laying out a general mathematical model and relating it to three strands in the literature on distributed algorithms.

> Define a *network transaction* to be the coordinated exchange of short messages between a collection of computers on an unreliable asynchronous network, where the message protocol is designed to satisfy a suitable SAFETY property and to TERMINATE with an answer in finite time.
>
> Assume that the computers are infinitely fast. Assume that the network has fixed latency and that it drops messages randomly.
>
> Measure the performance of the transaction by the time it takes to terminate, and by the probability that it terminates with the CORRECT answer. Study how these quantities depend on network latency, drop probability, retransmission strategy, and network topology.

(*a*) *'Distributed algorithms' model*

A distributed algorithm is an interconnected collection of autonomous computing nodes which communicate by exchanging messages. Many classic network algorithms, e.g. finding shortest paths, can be rewritten to consist of autonomous programs at each node, communicating along network edges, and running in slotted time—i.e. every clock tick, each node receives messages, then performs some local computation, then send messages. This is a reliable synchronous network.

In an asynchronous network, messages may take arbitrarily long but finite time to be delivered. There are many algorithms known for reliable asynchronous networks. Consider for example the problem of electing a leader from a network of several nodes. Each node is able to perform the local action Accept(me) to indicate that it considers itself leader. The leader election problem is to find an algorithm which satisfies

- LIVENESS: at any time, if there is no leader, a leader is eventually elected
- SAFETY: at any time, at most one node considers itself the leader

Some algorithms additionally provide

- TERMINATION: once a leader is elected, all nodes eventually enter a Terminated state in which they know whether or not they are leader and their decision cannot change

The focus is usually on finding algorithms which perform well, as measured by the total number of messages sent, and the total execution time (assuming some finite upper bound on message delivery time). Results are usually asymptotic in n, the number of nodes. For example, Villadangos et al. [17] give a clever algorithm which takes $O(n)$ messages and $O(n)$ time to elect a leader, assuming there is a Hamiltonian cycle known to all nodes and that any pair of nodes can communicate directly with each other.

For asynchronous networks with unreliable nodes (which may crash and stay crashed forever), there has been elegant logical analysis which has mainly produced impossibility theorems. It is impossible to solve the leader problem in such networks, without some *deus*

ex machina means of detecting which nodes have crashed [13]. This is because a crashed node cannot be distinguished from a slow node.

(*b*) *'Distributed systems' approach*

Asynchronous unreliable networks like the Internet are what we have to work with, and the next best thing is to weaken the LIVENESS or SAFETY requirements.

A famous example of an algorithm without LIVENESS is Paxos, uncovered by Lamport [7]. This is an algorithm for achieving consensus between a number of nodes. It guarantees a SAFETY property and also TERMINATION, but not LIVENESS—there are scenarios in which progress is not guaranteed.

An example of an algorithm without SAFETY is the leadership election algorithm by Gupta et al. [5]. This satisfies LIVENESS and TERMINATION, but instead of SAFETY it provides

- CORRECTNESS: the probability that more than one Terminated node thinks itself leader is low

The analysis assumes a probabilistic model—a system with n nodes connected in an Erdős-Rényi random graph, in which packet drops are independent. The LIVENESS guarantee holds for fixed n, almost surely over all execution traces; for CORRECTNESS a $\liminf_{n\to\infty}$ bound is given on the probability of producing a correct answer by some fixed time.

A similar probabilistic approach was taken by Wischik et ibid. [19], which discusses further the relationship between the logical predicate '$\Diamond$eventually' and the probabilistic predicate 'almost surely'. This work considers the problem of rendezvous in an asynchronous network with message loss. Suppose there are two generals on the top of hills, and they communicate via messengers who may be intercepted by barbarians in the valley below. If both generals decide to attack then the barbarians will be crushed; if only one attacks then the barbarians will win. The algorithm satisfies the properties

- TERMINATION: eventually both generals learn either that they are both going to attack, or that one will not
- SAFETY: if a thinks that b is going to attack, then b is going to attack

(but it is well-known that they cannot guarantee to coordinate the *time* of their attack). The TERMINATION statement holds almost surely over all execution traces, and the SAFETY holds at every instant in time for every possible execution trace.

(*c*) *'Interacting systems' model*

The classic model of an epidemic is also a kind of distributed system. Consider for example the following epidemic/gossip algorithm for leader election. Suppose that each node has an ID, and that there is a total ordering of IDs. Let each node randomly sends to its neighbours the highest ID it has heard of so far, and suppose that messages are dropped randomly. This algorithm has the properties that, in each connected component of a network, almost surely

- LIVENESS: eventually all nodes agree on a single ID
- CORRECTNESS: this is the highest ID in the component

Most results for epidemics are asymptotics in n, the number of nodes in the network.

Vasudevan et al. [16] propose a more sophisticated gossip algorithm for electing a leader, intended for ad-hoc networks with mobile nodes. In their algorithm, connectivity may change as nodes move, and so each node periodically pings its neighbours to check if it should update

its idea of who the leader is. They consider the case of constantly moving nodes and use simulation to study the fraction of time for which a node has a leader. (They also prove logical LIVENESS and SAFETY properties under the assumption that there is some time after which the network has reliable delivery.)

4. Three network transactions

We now study three different network transactions. The three problems are all trivial, to a distributed algorithms theorist who is only interested in logical safety and liveness—but they are interesting enough to a probabilist who is trying to estimate performance.

We will consider unreliable asynchronous networks, since this is the most reasonable model for Internet communication. We will calculate how the completion time of the algorithm depends on network latency and packet drop probability p, in the limit as $p \to 0$, for networks with a fixed number of nodes n. By contrast, performance analysis in the literature mostly studies the limit as $n \to \infty$. This is not a useful way to look at the performance of say web transfers, in which there are $n = 2$ parties, the web browser and the web server. Some sort of limit seems needed to get tractable answers, and $p \to 0$ should produce reasonable approximations for the range of packet drop probabilities described in Section 2. Some previous work on other problems, e.g. the Fast Byzantine Paxos algorithm of Martin and Alvisi [9], has considered completion time for arbitrary n, in the 'common case' where $p = 0$; we will calculate the common-case completion time and additionally the first-order dependency on p.

We will not be interested in calculating the number of messages sent, since as argued in Section 2 these are short messages which constitute an insignificant fraction of network traffic.

(*a*) *Retrieving a web page: http 1.0, 1.1, and 1.1 with pipelining*

Consider first a simple idealized web browser and server and network retrieving the `gmail.com` inbox. The web browser sends out request messages (pure requests, without the handshaking steps 1 and 2), and the server replies when it receives a request. The web browser sends out parallel requests for whatever items it needs next, as soon as it can. The web browser uses an exponential timer with mean RTO, which is constant, and the server does not use any timeouts—it is purely passive. Each message may be dropped with probability p, and successive drops are independent. The round trip time is RTT. It is straightforward to set up a Markov process to model the progress of the transaction, and to solve it with computer algebra, but this does not lead anywhere.

Instead, let us consider $\mathbb{E}T$, the expected time until the entire page has been retrieved, as a function of $q = 2p - p^2$, the round-trip packet drop probability, and let the retransmit timer be non-random. Now we will condition on the whether or not certain packets get dropped. Let A be the event that the first request+response for each item is not dropped, and let B_i be the event that the first request+response for each item apart from i is dropped, that the first request+response for i is dropped but the second is not. So $\mathbb{P}(A) = (1-q)^{66}$ since there are 66 items in total, and $\mathbb{P}(B_i) = q(1-q)^{66}$ and $\mathbb{P}(\text{neither } A \text{ nor any } B_i) = O(q^2)$. Then

$$\begin{aligned}\mathbb{E}T &= \mathbb{E}(T|A)\mathbb{P}(A) + \sum_i \mathbb{E}(T|B_i)\mathbb{P}(B_i) + O(q^2) \\ &= \mathbb{E}(T|A) + q\sum_i \Big[\mathbb{E}(T|B_i) - \mathbb{E}(T|A)\Big] + O(q^2)\end{aligned}$$

$$= 11\mathsf{RTT} + 12q\mathsf{RTO} + O(q^2).$$

The $O(1)$ term is the completion time in the common case, i.e. assuming no drops. The $O(q)$ term is the sum of additional completion times due to a possible drop, i.e. RTO times the number of items on the 'critical path'. We can straightforwardly apply this sort of reasoning to a fuller model of http, of which there are three versions (see Table 2), taking account also of whether or not the cacheable items have been cached. We have ignored server-side timeouts in this table, for convenience, and assumed that all requests go to the same web server.

uncached	idealized http	$\mathbb{E}T = 11\mathsf{RTT} + 12q\mathsf{RTO} + O(q^2)$
	http 1.0	$\mathbb{E}T = 50\mathsf{RTT} + 30q\mathsf{RTO} + O(q^2)$
	http 1.1	$\mathbb{E}T = 41\mathsf{RTT} + 17q\mathsf{RTO} + O(q^2)$
	http 1.1 pipelining	$\mathbb{E}T = 21\mathsf{RTT} + 17q\mathsf{RTO} + O(q^2)$
cached	http 1.0	$\mathbb{E}T = 20\mathsf{RTT} + 20q\mathsf{RTO} + O(q^2)$
	http 1.1	$\mathbb{E}T = 14\mathsf{RTT} + 20q\mathsf{RTO} + O(q^2)$
	http 1.1 pipelining	$\mathbb{E}T = 12\mathsf{RTT} + 12q\mathsf{RTO} + O(q^2)$

When the $O(q)$ term is equal to the $O(1)$ term, as with http 1.1 with pipeling and a cache, it indicates that the protocol does not impose any extra bottlenecks.

Some of the $O(q)$ terms here are very sensitive to discretization effects relating to how items are distributed between connections. It would be useful to consider also the expected completion time when the connection tree itself is random, to smooth away these discretization effects.

(*b*) *End-to-end versus hop-by-hop reliable delivery*

Here is a toy model of multihop transmission. Consider the problem of reliably delivering a message from a source node 0 to a destination node n across a series of intermediate nodes, where each link has its own packet drop probability p_i and round trip time RTT_i.

$$0 \xleftrightarrow{p_1, \mathsf{RTT}_1} 1 \xleftrightarrow{p_2, \mathsf{RTT}_2} \cdots \xleftrightarrow{p_n, \mathsf{RTT}_n} n$$

Hop-by-hop reliable delivery. When node i first hears the message it starts sending it to node $i+1$, resending it every RTO_{i+1} until it hears an acknowledgement from node $i+1$. Whenever node i hears the message it sends an acknowledgement back to node $i-1$.

End-to-end reliable delivery. Node 0 sends the message out, resending it every RTO until it hears an acknowledgement. The intermediate nodes are dumb relays: they just relay messages forwards and acknowledgements back. Node n sends back an acknowledgement whenever it hears the message.

† The number four was chosen by Netscape in the early days of web browsers. The unofficial FAQ `www.ufaq.org` says this is an appropriate number for users with slow modems. The reckoning might have been as follows. Consider a user with a 56kb/s modem, which has a packet size of 576 bytes, with four simultaneous connections to a server with an RTT of 250ms. The average window size for a connection is 0.76 packets. Perhaps one of the connections will be waiting for a reply, giving the others an average window size of 1.0 packets. This is just at the threshold of what TCP's fast recovery mechanism can cope with; any less and it will suffer frequent timeouts.

This rationale is clearly not appropriate for short messages, nor for broadband connection speeds.

http 1.0 uses a new TCP connection for every item, i.e. it goes through the full 6 steps listed in Section 1. A web browser can open multiple simultaneous connections, but it allows no more than four†of them to be in steps 1–5.

http 1.1 lets a single TCP connection handle multiple items in sequence. This means that steps 3 and 4 are replaced by

Client	*Server*
3a. Please send X_1.	
4a.	*Here's* X_1.
3b. Please send X_2.	
4b.	*Here's* X_2.

and so on as many times as needed. The protocol for handing the *Goodbye!* message is slightly different. RFC2616 recommends that no more than two simultaneous connections should be opened to a web server.

http 1.1 with pipelining allows requests to be pipelined. If a web browser has many requests ready, it can send them together rather than in sequence:

Client	*Server*
3a. Please send X_1.	
3b. Please send X_2.	
4a.	*Here's* X_1.
4b.	*Here's* X_2.

The default in Firefox 2.0.0.8 is not to use pipelining, and if it is turned on then to permit up to four pipelined requests. Internet Explorer 7 does not permit pipelining at all.

Table 2: The flavours of http

Let T be the time until node n receives the message, and suppose we wish to control $\mathbb{E}T$. Obviously, by making the retransmission timeouts small enough, we can with either scheme get $\mathbb{E}T = t^{\min} + \varepsilon$ where $t^{\min}$ is the propagation delay from node 0 to n, for any $\varepsilon > 0$.

There may be however be constraints on how small the retransmission timeouts can be. We have already seen that there is little point having these timeouts less than say $\approx$20ms because of correlations in packet drops. We will consider here a different constraint: computational burden. The relay nodes $1,\ldots,n-1$ may be handling many different messages, and in the hop-by-hop scheme a relay node will need to remember timeouts for each of these messages. It will need to (i) set a timeout whenever it forwards a message, (ii) cancel the timeout whenever it receives an acknowledgement, (iii) when it executes a timeout, work out the next timeout due to expire, and possibly (iv) when it receives a message, work out if it has already forwarded it. Operations (i)–(iii) typically use a data structure called a heap, and their complexity is $O(\log m)$ where m is the number of outstanding timeouts. Operation (iv) can be done at low cost using a hash table.

Consider the problem of choosing timeouts so as to minimize computational burden, subject to the constraint that $\mathbb{E}T = t^{\min} + \varepsilon$, and compare hop-by-hop to end-to-end. To make the working easier, suppose that acknowledgements are always reliably delivered.

Consider first a single link with drop probability p and round trip time RTT, and let $q = 1 - p$ and let RTO be the timeout. The expected delay until the message is received is

$$\mathbb{E}T = t^{\min} + \frac{1}{q} + (1+q)\frac{\mathsf{RTT}}{\mathsf{RTO}}.$$

The expected computational cost, measured as number of timers set plus number of timers processed plus number of acknowledgements processed, is

$$\mathbb{E}C = \frac{1-q}{q} + \Big\lfloor \frac{\mathsf{RTT}}{\mathsf{RTO}} \Big\rfloor + \Big(q \Big\lfloor \frac{\mathsf{RTT}}{\mathsf{RTO}} \Big\rfloor + 1 \Big).$$

Ignore the $\lfloor \cdot \rfloor$ for simplicity. The optimum for end-to-end is to choose RTO so as to

$$\text{minimize} \quad \frac{1}{\prod q_i} + \frac{(1+\prod q_i)\sum \mathsf{RTT}_i}{\mathsf{RTO}} \quad \text{such that} \quad \mathsf{RTO} \frac{1-\prod q_i}{\prod q_i} = \varepsilon,$$

and the minimal computational cost is

$$\frac{1}{\prod q_i} + \frac{1}{\varepsilon}\Big(\frac{1-(\prod q_i)^2}{\prod q_i} \sum \mathsf{RTT}_i \Big).$$

The optimum for hop-by-hop is to choose RTO_i so as to

$$\text{minimize} \quad \sum \frac{1}{q_i} + \sum \frac{(1+q_i)\mathsf{RTT}_i}{\mathsf{RTO}_i} \quad \text{such that} \quad \sum \mathsf{RTO}_i \frac{1-q_i}{q_i} = \varepsilon,$$

and the minimal computational cost is

$$\sum \frac{1}{q_i} + \frac{1}{\varepsilon}\Big(\sum \sqrt{\frac{1-q_i^2}{q_i} \mathsf{RTT}_i} \Big)^2.$$

Suppose for the sake of argument that all links have low drop probability $p_i \approx 0$ apart from one link which has drop probability p^*. Let this congested link have round trip time RTT^*. After some algebra, we find that end-to-end is better than hop-by-hop when

$$p^* < 1 - \frac{\sqrt{A^2+4} - A}{2} \quad \text{where} \quad A = \frac{\varepsilon(N-1)}{\mathsf{RTT} - \mathsf{RTT}^*}.$$

The critical threshold for p^* is an increasing function of A.

This result fits in with intuition and practice—if you have a local wireless link with high drop rates, it makes sense for the base station and client computer to recover quickly by using their own retransmissions and short timeouts; but in the wired Internet which has fewer drops it doesn't make sense to burden the routers with extra work.

(c) *Leader election*

Consider the following problem. A number of machines are connected to a hub, by access paths of different latencies. Any message sent by one machine to the hub will be broadcast to all the other machines. Messages may be dropped, either going to or from the hub; the drop probability is p and drops are independent. The problem is to elect a leader. A machine can perform the action $\mathsf{Accept}(X)$ to indicate that it accepts machine X as leader, and this is a TERMINATION action, i.e. it can perform Accept no more than once. We want an election algorithm which satisfies this SAFETY property: if X has done $\mathsf{Accept}(Y)$ then Y has done $\mathsf{Accept}(Y)$. The CORRECTNESS metric is the number of nodes who have Accepted themselves as leader—we want this to be close to one, but it is impossible to guarantee exactly one because of the chance of packet drops.

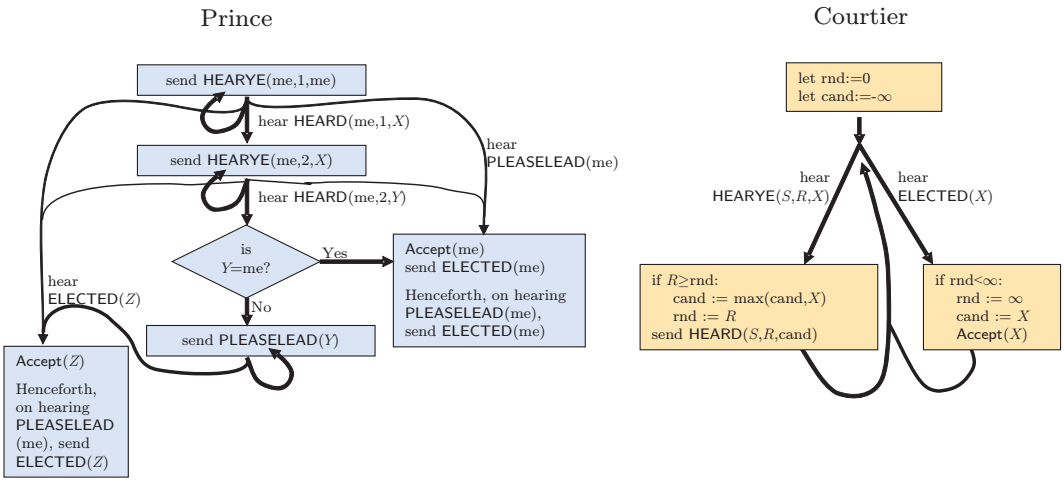

Figure 2: Prince and Courtier algorithms. The arrows show transitions from state to state. Some arrows are labelled, and these are transitions which are triggered by receipt of a message. The Prince has three unlabelled arrows, which mean: if none of the labelled transitions have been triggered within time RTO, then resend the message and reset the timeout.

This might be a simple model for the browser service in Microsoft Windows up to XP. The machines on an ad hoc wireless network elect one of their number to be the Master Browser, and this machine maintains a directory of the other machines and their printers and other facilities, which can be browsed in "My network neighbourhood". The hub represents the wireless shared access medium. The latencies might reflect how frequently a machine checks its message queue. The safety condition means that if a node queries its leader then the leader is at least expecting to receive queries.

Figure 2 shows a simple algorithm. The algorithm uses two types of nodes, courtiers and princes. A prince is a candidate for leader, and a courtier simply recognizes leaders. Assume that there is at least one prince and at least one courtier. Assume that each prince has a unique identity which it calls me, and that there is a total ordering on identities. The general idea is that princes broadcast who they are, in two rounds of broadcasts. The courtiers listen to the first round, then latch on to the best they've heard so far at the time the second round starts. The system however is asynchronous, and fast princes can start on the second round while slow princes are still on the first.

Section 3 described several other leader election algorithms, mostly more sophisticated than this simple algorithm. However, the simple algorithm and the assumption of a hub permit us to calculate performance measures, and I have not yet been able to do the same for the other algorithms.

Lemma 4.1 (Termination) *All Princes will eventually perform* Accept*, almost surely.*

Lemma 4.2 (Safety) *If X has done* Accept(Y) *then Y has done* Accept(Y)*.*

Lemma 4.3 (Correctness) *Let N be the number of leaders who are eventually elected. Assume that latencies are non-lattice. Then*

$$\mathbb{E}N \leq 1 + p + O(p^2)$$

for any set of start times for the princes.

Lemma 4.4 (Termination time) *Let t_P be the latency from the hub to the closest prince, and let t_C be the latency from the hub to the closest courtier. Suppose all princes start sending at the same time. Let T be the delay until an* ELECTED *message reaches the hub. Then*

$$\mathbb{E}T \leq 11t_P + 8t_C + 12p\mathsf{RTO} + O(p^2).$$

The algorithm could be modified to use more rounds. This would make it more robust but slower, i.e. decrease the $O(p)$ term in $\mathbb{E}N$, and increase the $O(1)$ term in $\mathbb{E}T$.

Proof of Lemma 4.1. All we need to check is that a prince cannot get stuck in the "send PLEASELEAD" state. Suppose the contrary, and let P be the prince with the highest ID among those who are eventually stuck. Suppose P wants Y to be the leader. This must be because P heard HEARD$(P, 2, Y)$, which was sent by a courtier in response to HEARYE$(P, 2, _)$. Suppose the courtier had state *rnd*, *cand* prior to receiving the HEARYE.

If $rnd = \infty$ then $cand = Y$ and the courtier has heard ELECTED(Y), so Y is in a state where it will eventually respond to P's PLEASELEAD(Y) messages.

If $rnd < \infty$ then $rnd \leq 2$ since that is the highest round number used in this algorithm, so by the courtier's algorithm $Y \geq P$. It cannot be that $Y = P$, otherwise P would not be asking Y to be leader. Therefore $Y > P$, so by assumption Y is not stuck in the "send PLEASELEAD" state, so Y is capable of responding to P and will eventually do so. □

Proof of Lemma 4.2. Safety is trivial, since X will only perform Accept(Y) on hearing ELECTED(Y), and the first ELECTED(Y) can only be sent by Y after it has done Accept(Y).□

Proof of Lemma 4.3. Each message has a name and a set of arguments. There are finitely many names and arguments; let $\mathcal{M}$ be the set of possibilities, and $M = |\mathcal{M}|$. Each message in $\mathcal{M}$ is transmitted on assorted links in assorted directions, and may be sent multiple times. Let $\mathcal{L}$ be the set of all links and directions, $L = |\mathcal{L}|$. The randomness consists in deciding for all messages $m \in \mathcal{M}$ and locations $l \in \mathcal{L}$ whether a specified copy $n \in \mathbb{N}$ is dropped at l.

Let A be the event that all messages in $\mathcal{M} \times \mathcal{L} \times \{1\}$ are delivered, so $\mathbb{P}(A) = (1-p)^{LM}$. We will use the synbol _ to denote an arbitrary value, and we will use words rather than notation to denote locations, so A can be written as the event that all messages $(\_, 1)$ are delivered. Let $B_{l,m}$ be the event that message $(m, 1)$ is dropped in location l but all other messages in $\mathcal{M} \times \{1, 2\}$ are delivered, so $\mathbb{P}(B_{l,m}) = p(1-p)^{2LM-1}$. Let C be the complement of $A \cup \bigcup_m B_m$, so $\mathbb{P}(C) = 1 - (1-p)^{LM} - LMp(1-p)^{2LM-1}$. Then

$$\begin{aligned}
\mathbb{E}N &= \mathbb{E}(N|A)\mathbb{P}(A) + \sum_{l,m} \mathbb{E}(N|B_{l,m})\mathbb{P}(B_{l,m}) + \mathbb{E}(N|C)\mathbb{P}(C) \\
&= \mathbb{E}(N|A)(1 - LMp) + p\sum_{l,m} \mathbb{E}(N|B_{l,m}) + O(p^2) \\
&= \mathbb{E}(N|A) + p\sum_{l,m} \Big(\mathbb{E}(N|B_{l,m}) - \mathbb{E}(N|A)\Big) + O(p^2). \qquad (4.1)
\end{aligned}$$

The $O(1)$ term is the common-case number of leaders elected, and the $O(p)$ term is the extra number of leaders elected as a consequence of single drops. Note that in calculating the second term we only need to consider l, m for which $\mathbb{E}(N|B_{l,m}) \neq \mathbb{E}(N|A)$.

Common case. Consider the random variable which is the state (*rnd*, *cand*) of each courtier and the state of each prince as a function of time. We argue first that conditional on

event A this random variable is uniquely determined. First, a courtier who receives $(m, 1)$ can never change state when it receives (m, i) for any $i \geq 2$. Second, a prince who has received a message $(m, 1)$ will either ignore $(m, 1)$ and all subsequent copies of m, or he will move to a state where (m, i) does not change his state for any $i \geq 2$. Putting these together, we see that no message (m, i) for $i \geq 2$ can change the state of any node, therefore it makes no difference whether or not any (m, i) is dropped.

Without loss of generality, consider the courtier C closest to the hub, and let $cand(t)$ and $rnd(t)$ be his state as a function of time. (All other courtiers have delayed versions of $cand(t)$ and $rnd(t)$.) Let t_2 be the first time that C receives a message $\mathsf{HEARYE}(\_, 2, \_)$, and suppose this message results in $cand(t_2+) = Q$. We will argue that $cand(t) = Q$ for all $t > t_2$.

First we prove that if C receives $\mathsf{HEARYE}(\_, 2, P)$ at $t_2 \leq t < t_\infty$, where t_∞ is the first time that C receives a message $\mathsf{ELECTED}$, then $P \leq Q$. We prove this by induction on the sequence of HEARYE messages received at or after time t_2. The claim is obviously true for the message received at $t = t_2$. Suppose that the claim is true for all messages prior to some time $t > t_2$. A $\mathsf{HEARYE}(\_, 2, P)$ at time t must have been triggered by a $\mathsf{HEARD}(\_, 1, P)$ sent out by C at time $u < t$. Either $u < t_2$, in which case the C's action at t_2 ensures $Q \geq P$. Or $u \geq t_2$, in which case the induction hypothesis ensures $P = Q$.

Now for the case of $t \geq t_\infty$. The $\mathsf{ELECTED}(P)$ message received at t_∞ was triggered by a $\mathsf{HEARD}(\_, 2, P)$ message at some time prior to t_∞, either directly or via a $\mathsf{PLEASELEAD}$. By the above, $P = Q$.

We have proved that, conditional on A, every courtier latches onto a unique prince Q, i.e. all $\mathsf{HEARD}(\_, 2, x)$ messages have $x = Q$. By looking at the prince algorithm, we see that only prince Q can be Accepted.

Single-drop case. Now condition on event $B_{l,m}$, i.e. suppose that a single message $(m, 1)$ is dropped in location l but all other messages in $\mathcal{M} \times \{1, 2\}$ are delivered. By a similar argument to the common case, conditional on $B_{l,m}$ all the states of all the nodes at all times are completely determined. We want to calculate for each l, m how many princes end up becoming a leader. In fact, a simpler way to approach the problem is to let Q be the elected leader in scenario A, and to ask which scenarios $B_{l,m}$ would result someone other than Q becoming a leader.

Pick some prince $P \neq Q$, and suppose P becomes a leader in scenario $B_{l,m}$. P must have become leader as a consequence of hearing either $\mathsf{HEARD}(P, 2, P)$ or $\mathsf{PLEASELEAD}(P)$, and we consider the two cases separately.

Suppose first that P becomes leader as a consequence of hearing $\mathsf{HEARD}(P, 2, P)$. It must previously have sent $\mathsf{HEARYE}(P, 1, P)$, and suppose this should have reached the courtier C nearest the hub at t_1^P. Define t_2^P and t_1^Q similarly. Note that t_1^P and t_1^Q are non-random—they depend only on the start times and latencies—whereas t_2^P depends on the particular scenario $B_{l,m}$. Now, if $t_2^P < t_1^Q$ in this scenario, then in scenario A it must also happen that C hears $\mathsf{HEARYE}(P, 2, \_)$ before it has heard $\mathsf{HEARYE}(Q, 1, Q)$, so Q will not be elected, which contradicts the choice of Q. Therefore $t_2^P > t_1^Q$. Then the only way that P can end up hearing $\mathsf{HEARD}(P, 2, P)$ is if (i) the message drop is to or from C, and (ii) the drop is of $\mathsf{HEARYE}(Q, 1, Q)$.

Suppose next that P becomes leader as a consequence of hearing $\mathsf{PLEASELEAD}(P)$. This must have been triggered by some other prince hearing $\mathsf{HEARD}(\_, 2, P)$, which was triggered by them sending $\mathsf{HEARYE}(\_, 2, \_)$. Let t_2 be the time at which this HEARYE should have reached courtier C. As in the first case, it must be that $t_2 > t_1^Q$, and again the message drop must have been of $\mathsf{HEARYE}(Q, 1, Q)$ on its way to C.

In fact we can narrow down the location of the message drop even further. Suppose that $\mathsf{HEARYE}(Q, 1, Q)$ never reaches C because it is dropped on its way to the hub. The same argument as for the common case shows that all courtiers latch onto a single prince, who is elected leader.

We have shown that if $B_{l,m}$ results in some prince other than Q becoming leader, then the dropped message (l, m) is $\mathsf{HEARYE}(Q, 1, Q)$ on its way from the hub to courtier C. This must be the only non-zero term in the sum in (4.1).

Finally, consider the number of nodes who might be elected leader as a consequence of this drop. It is not hard to see that the only princes who can possibly become leader are Q, plus the first prince from whom C hears $\mathsf{HEARYE}(\_, 2, \_)$. Therefore the only non-zero term in the sum in (4.1) is ≤ 1. □

Proof of Lemma 4.4. Let P be the closest prince, C the closest courtier, and Q the prince who is elected leader in the common case. Let t_Q be the latency from Q to the hub. As in the proof of correctness, the $O(1)$ term is the delay in the common case of no drops, and the $O(p)$ term is the extra delay due to single drops.

For the common case, when there are no drops, consider the chain of events

- time 0: P sends $\mathsf{HEARYE}(P, 1, P)$
- time $t_P + t_C$: C replies $\mathsf{HEARD}(P, 1, \_)$
- time $2t_P + 2t_C$: P sends $\mathsf{HEARYE}(P, 2, \_)$
- time $3t_P + 3t_C$: C replies $\mathsf{HEARD}(P, 2, Q)$
- time $4t_P + 4t_C$: P sends $\mathsf{PLEASELEAD}(Q)$
- time $5t_P + 4t_C + t_Q$: Q sends $\mathsf{ELECTED}(Q)$
- time $5t_P + 4t_C + 2t_Q$: $\mathsf{ELECTED}$ reaches the hub

The election may not actually happen this way, but there is no way it could happen any slower. For example, we know from the proof of the correctness lemma that C latches on to the eventual leader when it first receives a $\mathsf{HEARYE}(\_, 2, \_)$, and this must have happened by time $3t_P + 3t_C$ if Q is to become leader. Furthermore, it must have happened because C received a $\mathsf{HEARYE}(Q, 1, Q)$, therefore $t_Q + t_C \leq 3(t_P + t_C)$. Putting all this together, we find that in the common case, $T \leq 11t_P + 8t_C$.

In the single-drop case, we need to find the possible extra delay caused by any dropped packet. If all the twelve messages involved in the above chain of events are delivered, then there is no extra delay beyond $11t_P + 8t_C$. If one of them is dropped, the worst that can happen is an extra delay of RTO. This gives the $12p\mathsf{RTO}$ term. □

5. Conclusion

Ever more computer applications run over the Internet, and these applications perform ever more sophisticated tasks through interacting with other networked computers. For many applications it is good use of the network which gives value, not isolated computer power. This is the case not just for the Internet—it holds too for communication between cores of a multicore processor.

The trouble with networks is that the speed of light limits how fast these interactions can be, and unreliable networks which lose data will make the slowdown even worse. It's important to understand how the performance of algorithms depends on the network: its latency, its drop probability, and its topology.

Just as there is a standard set of algorithms and data structures for which we know how computation time and memory requirements depend on the size of the input data, just so

can we imagine a collection of distributed algorithms for which we know how performance depends on the characteristics of the network. Much is known already about how performance of certain algorithms depends on the size of the network, but little is known about how it depends on network latency and drop probability.

The most useful outcome of this research will be a better understanding of layering. Some network problems can be solved in multiple places—for example, reliable delivery is achieved by timeouts and retransmission at the wireless link layer built into the hardware of your wifi card, then at the TCP layer built into the operating system, then at the application layer in Google's Javascript code. It's not clear *a priori* at which layer reliability should be implemented, nor what sorts of channels are needed so that Google might communicate its reliability requirements to the wifi card. This paper has taken a small step, by giving a method of analysis which applies equally to all layers. The real win will be if we can find some sort of calculus which can tell us the net performance of a system composed of a number of distributed algorithms.

References

[1] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's initial window, 2002. URL `ftp://ftp.rfc-editor.org/in-notes/rfc3390.txt`. RFC 3390.

[2] Edith Cohen and Haim Kaplan. Prefetching the means for document transfer: a new approach for reducing web latency. *Computer Networks*, 2002. URL `http://www.research.att.com/~edith/publications.html`.

[3] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 1997. URL `http://doi.acm.org/10.1145/233013.233038`.

[4] John Doyle and J. M. Carlson. Power laws, highly optimized tolerance, and generalized source coding. *Physical Review Letters*, 2000. URL `http://www.cds.caltech.edu/~doyle/CmplxNets/`.

[5] Indranil Gupta, Robbert van Renesse, and Kenneth P. Birman. A probabilistically correct leader election protocol for large groups. In *Proceedings of 14th International Symposium on Distributed Computing (DISC)*, pages 89–103, 2000. URL `http://www.cs.cornell.edu/gupta/Papers/disc2000.final.pdf`.

[6] Van Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM*, 1988. URL `http://ee.lbl.gov/papers/congavoid.pdf`.

[7] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 1988. URL `http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos`.

[8] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 1994. URL `http://dx.doi.org/10.1109/90.282603`.

[9] J-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2005. URL `http://www.cs.utexas.edu/users/lasr/papers/Martin05Fast.pdf`.

[10] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet mathematics*, 2004. URL `http://www.eecs.harvard.edu/~michaelm/ListByYear.html`.

[11] V. Paxson and M. Allman. Computing TCP's retransmission timer, 2000. URL `ftp://ftp.rfc-editor.org/in-notes/rfc2988.txt`. RFC 2988.

[12] S. Rewaskar, J. Jaur, and F. D. Smith. A performance study of loss detection/recovery in real-world TCP implementations. In *Proceedings of IEEE ICNP*, 2007. URL `http://www.cs.unc.edu/~jasleen/research/tcp-analysis/`.

[13] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell, 1995. URL `http://hdl.handle.net/1813/7146`.

[14] A. S. Tanenbaum, J. N. Herder, and H. Bos. File size distribution in UNIX systems—then and now. *Operating System Review*, 2006. URL `http://www.cs.vu.nl/~ast/publications/`.

[15] Murad S. Taqqu, Walter Willinger, and Robert Sherman. Proof of a fundamental result in self-similar traffic modeling. *ACM/SIGCOMM Computer Communication Review*, 1997. URL `http://doi.acm.org/10.1145/263876.263879`.

[16] Sudarshan Vasudevan, Jim Kurose, and Don Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Proceedings of IEEE ICNP*, 2004. URL `http://doi.ieeecomputersociety.org/10.1109/ICNP.2004.1348124`.

[17] J. Villadangos, A. Córdoba, F. Fariña, and M. Prieto. Efficient leader election in complete networks. In *Proceedings of Euromicro-PDP*, 2005. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1386052`.

[18] A. Willig, M. Kubisch, C. Hoene, and A. Wolisz. Measurements of a wireless link in an industrial environment using an IEEE 802.11-compliant physical layer. *IEEE Transactions on Industrial Electronics*, 2002. URL `http://www.tkn.tu-berlin.de/~awillig/`.

[19] Lucian Wischik and Damon Wischik. A reliable protocol for synchronous rendezvous (note). Technical Report 2004-1, University of Bologna, 2004. URL `http://www.wischik.com/lu/research/verona.html`.

[20] Maya Yajnik, Sue Moon, Jum Kurose, and Don Towsley. Measurement and modelling of the temporal dependence in packet loss. In *Proceedings of IEEE INFOCOM*, 1999. URL `http://www-net.cs.umass.edu/networks/publications.html`.