# 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 transaction, 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, notably the fact of heavy-tailed distributions in traffic, 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, if I in London want to retrieve `www.nytimes.com` from New York then it should take 38ms, plus a handful of milliseconds for my computer to formulate the request and the webserver to formulate the reply, plus a few more milliseconds since the transatlantic cables will not be perfectly direct. In practice, it often takes at least four *seconds*. In the rest of this section I will outline where delay can arise. Cohen and Kaplan [2] give a more technical account.

**Anatomy of a web page.** When a web browser sends a request for, say, `www.nytimes.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 websites, 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

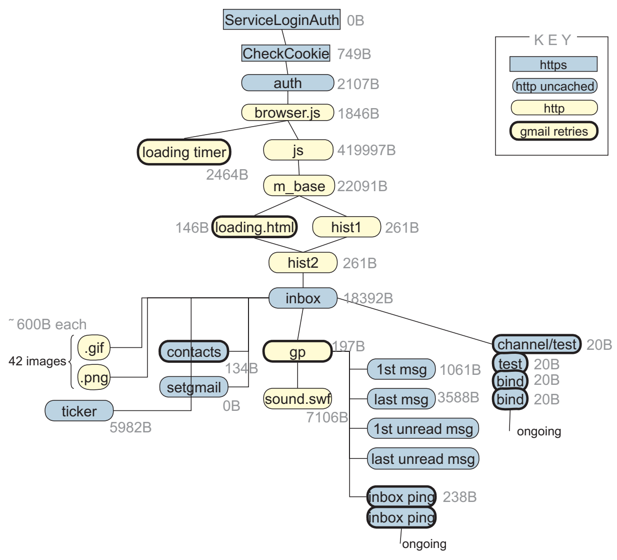

Figure 1: The graph of dependencies for retrieving an inbox at `gmail.com`, gathered on 28 August 2006 using Firefox 1.07. Paros 3.2.13 was used to trap each http request. On successive visits, different sets of requests were let through; the set of ensuing requests reveals the graph. The numbers show 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 Section 3a. All this back-and-forth leads to delay. To cut some of the 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 we 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!* |

The first two steps are there to prevent one round of messages from getting confused with another [13]. To retrieve an item over a secured connection (`https`), as for the first two items in Figure 1, there are some extra steps in between 2 and 3 which in the best case are

| | Client | *Server* |
|---|---|---|
| 2a. | Let's use cipher $c$. | |
| 2b. | | *Here's my public key.* |

| | | |
|---|---|---|
| 2c. | Here's a secret key for all further data. | |
| 2d. | | *Ready.* |

**Reliable delivery over an unreliable network.** The Internet is inherently unreliable and may drop messages *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 a message, then waits for an acknowledgement that it has been received (except for the final *OK!* which does not expect an acknowledgement). TCP sets a retransmission timeout $\mathsf{RTO}$, and if no acknowledgement is forthcoming within $\mathsf{RTO}$ then it retransmits. The acknowledgement is usually bundled with the reply message. Ideally $\mathsf{RTO}$ would be the round trip time from one party to the other and back again, but round trip time varies from packet to packet so $\mathsf{RTO}$ is set to take account of estimated mean and variability. Each side of the communication maintains its own copy of $\mathsf{RTO}$, which it updates continuously. The complete specifications of how $\mathsf{RTO}$ is updated are in [9], and a survey of how it is implemented in practice is given by Rewaskar et al. [10]. Broadly speaking, if the round trip time experienced by a packet is $\mathsf{RTT}$, then

(i) $\mathsf{RTO}$=3 seconds for the first in a round of messages
(ii) $\mathsf{RTO}=\max(3\mathsf{RTT}, 200\text{ms})$ for the second message
(iii) $\mathsf{RTO}$ 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) When there are many losses, a given message may be retransmitted multiple times. $\mathsf{RTO}$ is used as the timeout the first time, $2\mathsf{RTO}$ the second time, $4\mathsf{RTO}$ the next, and so on.

The above description holds for short messages. Larger messages are broken into packets (typical packet sizes are 1500bytes for ethernet and 576bytes for a modem) and the packets are reliably delivered. Messages 1 and 2 always fit into one packet. If message 3 is several packets long, TCP sends the first packet, then waits for it to be acknowledged using the $\mathsf{RTO}$ mechanism; once acknowledged, TCP 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. The last packet of a message always uses the $\mathsf{RTO}$ mechanism.

There are 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 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 the first file in directory $d$? | |
| 4. | | *It is* $f_1$. |
| 5. | What is the next file after $f_1$? | |
| 6. | | *It is* $f_2$. |
| | $\vdots$ | |
| $n+4$. | 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, like Facebook pages full of widgets, 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

When a packet is dropped, the RTO timeout is at least 200ms and can be as much as 3 seconds. 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 is dropped there's a good chance another copy will make it through. This would make TCP 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 [5]. That experience resulted in the current design of TCP, and a strong disinclination to experiment with making TCP more aggressive. In this section I will argue that the statistics 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 (note that many of the items in the `gmail.com` inbox are this small). Of course there is no point sending the copies so close together that they share the same fate.

There is incidentally another slowdown in TCP, which occurs when it sends messages more than one packet long: it waits for the other party to acknowledge the first packet before sending the rest. There has been a proposal that TCP should be allowed to send up to four packets or so without waiting for an acknowledgement [1]. This would save at least one round trip time. It would also give TCP's rapid retransmit mechanism a chance to work, though the last packet in a message would still be subject to the RTO timeout. This scheme is widely implemented, but it seems to be turned off by default.

**The central paradox of communication network design:** 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.

*What do users value?* The 3 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, although that also includes the cost of storing the message until it can be delivered).

*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 [8] 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 from 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 over long timescales) [6]. Researchers went looking for the cause, and proved that the aggregate of many heavy-tailed flows will lead to self-similarity and long-range dependence [12], whereas light-tailed flows will not. Long-range dependence is important for network engineering since it means that networks need very large buffers; and self-similarity is clearly visible in plots of network load; hence the interest of network engineers in heavy tails.

*How much traffic is made up of elephants?* Tanenbaum et al. [11] has collected datasets which let 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 the Computer Science filestore at the Vrije Universiteit, measured in 1984 and 2005. (I have used this data on file sizes, rather than Internet traffic statistics, because I have not found historical Internet traffic data which gives sufficiently fine detail about the distribution of small message sizes. Internet traffic engineers have been more interested in elephants than in mice.) 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.

**Packet drop statistics.** Packet drop rates vary wildly, by time and location, and congestion hotspots hop around the network. The snapshots at `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 [17] 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 [14] 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.)

**Conclusion.** Whereas network engineers concentrate on the elephants, for our purposes the mice are more interesting. In networks which are provisioned to carry elephants, that is to say *any* general-purpose network, 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. 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. Three network transactions

We now study three different network tasks. We will calculate how the completion time of the task depends on the network latency, the retransmit timeout, and the packet drop probability $p$, in the limit as $p \to 0$.

Performance analysis in the literature has mostly been concerned with the limit $n \to \infty$, where $n$ is the number of nodes, and has usually assumed $p = 0$. 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. The literature has mostly been concerned with the number of messages sent, but as argued in Section 2 these are short messages which constitute an insignificant fraction of network traffic. A fuller review of the literature, as well as detailed calculations and proofs of the following results, can be found in the extended version of this paper [15].

### (*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. 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 a fixed retransmit timeout RTO, 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. We shall calculate $\mathbb{E}T$, the expected time until the entire page has been retrieved, as a function of $q = 2p - p^2$, the round-trip packet drop probability. By conditioning on which if any packets are dropped, one can show that for the `gmail.com` web page,

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

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'.

**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. 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 1.1** 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.

Figure 2: The flavours of http

We can straightforwardly apply this sort of reasoning to a fuller model of http, of which there are three versions (see Figure 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 pipelining 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 dependency graph 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. We will consider two algorithms for 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 $\mathsf{RTT}_i$.

$$0 \xleftrightarrow{p_1, \mathsf{RTT}_1} 1 \xleftrightarrow{p_2, \mathsf{RTT}_2} 2 \cdots (n-1) \xleftrightarrow{p_n, \mathsf{RTT}_n} n$$

One algorithm is *hop-by-hop reliable delivery.* When node $i$ first hears the message it starts sending it to node $i+1$, resending it every $\mathsf{RTO}_{i+1}$ until it hears an acknowledgement from node $i+1$. Whenever node $i$ hears the message it sends an acknowledgement back to node $i-1$. Another algorithm is *end-to-end reliable delivery.* Node 0 sends the message out, resending it every $\mathsf{RTO}$ 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.

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 arbitrarily small $\varepsilon > 0$. There may 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 $\mathbb{E}T = t^{\min} + \varepsilon$. To make the working easier, suppose that acknowledgements are always reliably delivered. Then it is possible to calculate explicitly the computational burden for hop-by-hop versus end-to-end, measured as number of timers set plus number of timers processed plus number of acknowledgements processed; detailed calculations are in [15]. 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 $\mathsf{RTT}^*$. After some algebra, we find that end-to-end is better than hop-by-hop when

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

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. This might be a simple model for the browser service in Microsoft Windows up to XP—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, and the latencies might reflect how frequently a machine checks its message queue.

Formally, we suppose that each machine can perform the action $\mathsf{Accept}(X)$ to indicate that it accepts machine $X$ as leader. We will require that this be a terminal decision, i.e. no machine can perform $\mathsf{Accept}$ more than once; and we will be interested in how long it takes for all machines to terminate. The leader election problem introduces a new issue: unless all machines know each other in the first place (which makes the problem pointless, since they might as well pre-arrange to select the highest-numbered machine), it is impossible to guarantee that the algorithm will terminate with a single leader. This is because there may be machines between which all messages happen to be dropped—so some machines either terminate with the wrong leader, or never terminate at all. To acknowledge this, we will seek to meet a hierarchy of successively weaker probabilistic requirements.

(i) If $X$ has done $\mathsf{Accept}(Y)$ then $Y$ must have done $\mathsf{Accept}(Y)$. Such a condition is known in the literature as safety. We want this to be logically true, i.e. true for every point in the sample space and not just 'almost surely', in case we have the wrong probability model for packet drops. This property ensures the network cannot enter an inconsistent state.

(ii) Every machine should terminate. This is known as liveness. We want this to be almost surely true, but it is unreasonable to insist that it be logically true given that our probability model has independent drops.

(iii) Let $N$ be the total number of leaders elected. We want $\mathbb{E}N$ to be not much larger than 1. If $p = 0$ (also referred to as the 'common case' or 'graceful execution') we want $N = 1$ almost surely.

(iv) We wish to know the expected time until all machines have terminated.

The literature on distributed algorithms e.g. the book by Lynch [7] has concentrated on logical truth. The extended version of this paper outlines other approaches, including the link between logical and almost-sure truth described by Wischik et idem [16].

Figure 3 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 $\mathsf{me}$, 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

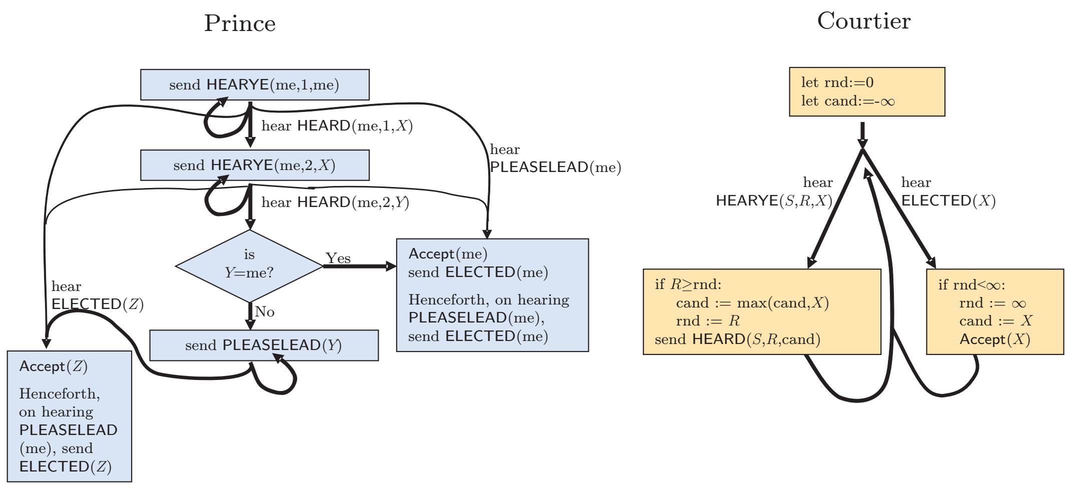
Prince
send HEARYE(me,1,me)
hear HEARD(me,1,X)
send HEARYE(me,2,X)
hear HEARD(me,2,Y)
hear PLEASELEAD(me)
is Y=me?
Yes
No
Accept(me) send ELECTED(me) Henceforth, on hearing PLEASELEAD(me), send ELECTED(me)
hear ELECTED(Z)
send PLEASELEAD(Y)
Accept(Z) Henceforth, on hearing PLEASELEAD (me), send ELECTED(Z)
Courtier
let rnd:=0
let cand:=-∞
hear HEARYE(S,R,X)
hear ELECTED(X)
if R≥rnd: cand := max(cand,X) rnd := R send HEARD(S,R,cand)
if rnd<∞: rnd := ∞ cand := X Accept(X)

Figure 3: 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.

start on the second round while slow princes are still on the first. The extended version of this paper describes 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. The following theorem is proved in [15].

**Theorem** *The algorithm satisfies the safety and termination conditions (i) and (ii).*

*Let $N$ be the number of leaders who are eventually elected. Then $\mathbb{E}N \le 1 + p + O(p^2)$, assuming that there is no linear combination of latencies which is equal to 0.*

*Let $T$ be the delay until an ELECTED message reaches the hub. Let $t_P$ be the latency from the hub to the closest prince, let $t_C$ be the latency from the hub to the closest courtier, and assume all princes start sending at the same time. Then $\mathbb{E}T \le 11t_P + 8t_C + 12p\mathsf{RTO} + O(p^2)$.*

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 $\mathbb{E}T$.

## 4. 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 and and memory caches on a multicore processor.

The trouble with networks is that there is delay inherent in every interaction, because of the speed of light; in unreliable networks the delay is exacerbated by the need for retransmissions. So it is important to understand how the performance of a networked application depends on the underlying network. Computer science has developed a reportoire of efficient algorithms and data structures for standalone computers, and the discipline of complexity analysis which studies how computation time and memory requirements depend on the size of the input data. In just the same way, we need a discipline of network complexity analysis, which studies how execution time and correctness depend on the network's latency, its drop probability, and its topology.

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 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 overall 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. RFC 3390.

[2] Edith Cohen and Haim Kaplan. Prefetching the means for document transfer: a new approach for reducing web latency. *Computer Networks*, 2002.

[3] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 1997.

[4] John Doyle and J. M. Carlson. Power laws, highly optimized tolerance, and generalized source coding. *Physical Review Letters*, 2000.

[5] Van Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM*, 1988.

[6] 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.

[7] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[8] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet mathematics*, 2004.

[9] V. Paxson and M. Allman. Computing TCP's retransmission timer, 2000. RFC 2988.

[10] 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.

[11] A. S. Tanenbaum, J. N. Herder, and H. Bos. File size distribution in UNIX systems—then and now. *Operating System Review*, 2006.

[12] Murad S. Taqqu, Walter Willinger, and Robert Sherman. Proof of a fundamental result in self-similar traffic modeling. *ACM/SIGCOMM Computer Communication Review*, 1997.

[13] Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 1987.

[14] 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.

[15] Damon Wischik. Short messages (long version), 2008. URL `http://www.cs.ucl.ac.uk/staff/d.wischik/Research/shortmsg.html`.

[16] Lucian Wischik and Damon Wischik. A reliable protocol for synchronous rendezvous (note). Technical Report 2004-1, University of Bologna, 2004.

[17] 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.