Dartmouth College • Department of Computer Science • Senior Thesis

Admission Control Policies for Internet File Transfer Protocols

Simon Holmes à Court
Advisor: John Danskin
TR#: PCS-TR97-315
30 May 1997

Abstract

Server congestion is a major cause of frustration with the Internet. It is not uncommon for a server with a new release of popular software to be swamped by many times more clients than it can possibly handle. Current Internet file transfer protocols, namely FTP and HTTP, do not have any policy to regulate client admission.

In this thesis we are concerned with server admission policies that will improve clients' experience with servers under heavy load. Using a purpose-built network simulator, we compare the prevalent protocols with two new protocols that include policies taken from processor scheduling.

By applying more intelligent client admission policies it is hoped that the quality of service on the Internet can be improved.

Introduction

When a heavily anticipated software release is posted on the Internet a downloading frenzy begins. Thousands of clients attempt to download a small set of files from a small number of servers all at once. Many will be repeatedly turned away as the server is unable to handle the peak load. Although the Internet is designed for many-to-many communications, at times like these, at a micro level, a subset of it behaves as a many-to-one system.

Demand may peak for a few hours or days at most, and then the server will return to a much reduced activity level. However during the heavy load period users can spend long periods trying to initiate downloads, and when they finally get through, server performance can be excruciatingly poor. Under such heavy load, a significant proportion of users will be greatly inconvenienced. As the commercial world embraces the Internet software distribution model, network clients are increasingly customers. Bad experiences with a corporation's server affect its on-line image. Even for non-commercial service providers, providing the client with the highest possible quality of service should be of paramount concern.

All servers in an overload state must degrade service for at least some of their clients. The key to maximising client utility is having performance degrade gracefully under load. Using current protocols, we can show that doubling the load on a server can increase client session times by over

1000%. On the Internet, even modest file transfers can take minutes. When a human is waiting for the transfer to finish, large blow-outs in service times must be avoided at all costs.

We are concerned with the use of policy to maximise client utility. By implementing more intelligent client admission policy it is hoped that the quality of service offered by the server can be improved. Client admission policy is largely seen as a scheduling problem—how to best schedule the servicing of arriving file transfer requests.

In this thesis two protocols based on new job scheduling policies are developed. One of the new policies applies the principle of shortest-job-first, the other applies the principle of first-come-first served. These policies can be readily added to the HTTP protocol. Our experiments use a purpose-built network simulator to compare the two new protocols against the performance of the prevalent Internet transfer protocols, HTTP and FTP. Specifically, we are interested in relative client utility under high-load server conditions.

Client Utility

Maximising client utility should be the primary concern of the server—server measures such as throughput and efficiency amount to little if a substantial number of users are dissatisfied with their interactions with the server. There are no obvious measures of client utility, but the following principles are factors relevant to addressing the issue:

- As much as possible, the server should minimise the total time between a client's initial connection attempt and the successful termination of the requested transfers.
- Jobs should be admitted to the server in a near-first-come-first-served basis to ensure fairness. It is not crucial that jobs be admitted in a strict order, rather, we want to ensure that the first client requesting a given resource is served before any other client requesting the same resource.
- If the server is busy, it is preferable to acknowledge a new client's request and confirm that the resources it requests are available than to simply turn the client away expecting it to retry at a later stage.

The Problem from a Scheduling Perspective

Maximising client utility is essentially a problem of optimal file transfer scheduling. File transfer scheduling is related to processor scheduling problems. While we can draw on some general principles

of processor scheduling, the client/server network model introduces many important differences that need to be addressed.

Scheduling theory dictates that optimum scheduling is obtained under shortest-job-first policy. Typically, however, the scheduler cannot know in advance the duration of the job. In file transfer scheduling the server can use the requested file's size as a guide to job duration. The client's connection speed is part of the equation, however, and connection speeds on the Internet may vary by at least an order of magnitude. It is generally not possible to know the speed of a TCP connection until the transfer session is well under way, so the best we can do is assume that regardless of connection speed very small transfers will invariably take less time than very large transfers.

In processor scheduling, the number of concurrent processes is typically limited by the size of the process table. The size of the table is set such that under normal use the user will rarely run up against the bounds of the table. In file transfer scheduling, the number of concurrent jobs is limited by the number of network sockets allocated to the server. Under high loads, it is common for all the server's sockets to be in use, and many would-be clients must be turned away.

When a processor scheduler is unable to immediately schedule a new job, a possible strategy is to hold the job until the scheduler is ready to begin processing the job. When a client requests a file from a server that can not handle the job immediately, holding onto the client would consume a needed server socket. A possible strategy is to ask the client to return at a later time—however there is no guarantee that the client will return at exactly the requested time, if at all. If the client connects to the Internet via a dial-up connection, there is a significant likelihood that the client will not be around at any server specified future time to attempt reconnection.

When a server is utilising all available sockets and must turn away a client, we can expect the unsatisfied client to repeatedly retry for a connection. Eventually, in the period between a client's connection attempts, a server socket will free up. Unless there is a policy to specifically address this issue, the next client that attempts connection will take the socket, regardless of how long it, or others, have been waiting. If a client has just disconnected from the server and wishes to reconnect, it will often be able to reclaim its socket before any waiting clients attempt to connect. Without appropriate policy, clients will not be fairly or intelligently admitted.

In order to maximise the clients' utility, a policy needs to address file transfer job scheduling. While some principles of processor scheduling apply to file transfer scheduling, the distributed and unpredictable nature of internet file transfers introduce new challenges.

Previous Work

Two protocols, FTP and HTTP, are responsible for the lion's share of Internet file downloads. Neither protocol adequately addresses the above issues.

FTP

The File Transfer Protocol[1] (FTP), the mainstay of Internet file transfers for the past 26 years, is an interactive, session-based protocol. The client initiates a TCP connection to the server and maintains the connection until it chooses to disconnect, or the server disconnects after a typically generous timeout. As long as the client is connected, a server socket is dedicated to the client, even if the client is idle. When a human is interacting with the server, there will likely be significant periods of idle time.

The interactive portion of an FTP session is conducted over the 'control' connection. When the client requests a file, another network connection, the 'data' connection, is opened, the file is transferred and the connection is closed. The dual-port requirement of FTP adds significant overhead to the protocol.

Increasingly GUI-based FTP clients are used to handle the connection for the user. A typical connection transcript (from the popular Anarchie FTP client) follows. Entries starting with 'S' and 'C' are the server and client respectively:

```
S: Welcome
C: USER anonymous
S: 331 Guest login ok, send your complete e-mail address as password.
C: PASS *****
S: 230 Welcome to the Dartmouth College Anonymous FTP Server.
C: PWD
S: "/" is current directory.
C: MACB E
S: 500 'MACB E': command not understood.
C: TYPE A
S: 200 Type set to A.
C: SIZE /pub/README
S: 213 1963
C: PORT 129,170,40,111,8,5
S: 200 PORT command successful.
C: RETR /pub/README
S: 150 Opening ASCII mode data connection for /pub/README (1911 bytes).
S: 226 Transfer complete.
C: BYE
```

To transfer a 2K file, 19 messages were sent between client and server on the control connection, a 'data connection' was established, 2K of data transferred and both connections closed. Clearly, FTP imposes a overhead that is significant whenever latency is high or files are small.

Most FTP servers place an upper limit on the number of concurrent users. While the server is working at the limit, all newly arriving clients are turned away, even if the current users are all idle. Repeated reconnection attempts are the only option open to the rejected client.

The protocol has no policy to ensure that fast connections are dealt with first, or that rejected clients are eventually dealt with in a fair order. FTP's lack of scheduling policy, relatively high overhead for small files, and recognised security drawbacks[2] should be reason enough to revamp or abandon the protocol. Perhaps the protocol's only real strength is its ubiquity. FTP's inadequacies are partly addressed by HTTP.

HTTP

At the time of writing the most prevalent implementation of HTTP is v1.0.[3] The protocol was designed as a generic data transfer protocol, and is the backbone protocol of the World Wide Web. Clients open a TCP connection to the server and immediately submit a request message. If there are no security restrictions on the entity there is no further preamble and the server replies with the requested entity. The end of file is signalled by closing the network connection. As such, under version 1.0, HTTP requires a separate connection for each file requested.

HTTP was not designed to transfer large entities. Originally the protocol was used almost exclusively for small transfers (HTML documents and JPEG/GIF image files), however there is a trend towards utilising HTTP for all file transfers, in place of FTP; many web sites now offer file transfers through HTTP in preference to FTP.

There are no policies within the protocol that address scheduling. As with FTP, when the server's sockets are fully committed, there is no policy to ensure that waiting clients will be admitted fairly. The original design goals of speed and simplicity are undermined in the not uncommon case of a busy server.

HTTP version 1.1[4] allows for 'persistent connections' whereby the client can request and receive multiple entities in the course of a single network connection. This spreads the cost of opening a TCP connection and the TCP ramp up time incurred in the one-entity-per-connection model. Despite the Internet Engineering Task Force's ongoing work to improve the protocol, HTTP v1.1 does not address the scheduling issues outlined above.

A 10 MB file takes approximately 50 minutes to download over a 28.8 Kbps modem—the reality for a large proportion of the Internet. Even though Internet connection speeds are expected to improve

significantly in the near future, growth in the sizes of typical transfers can be expected to diminish any real improvement. We can expect long file transfers to be a fact of Internet life for some time to come. As long as there are long file transfers, limited server sockets and no scheduling policies, long jobs will block short jobs. Neither FTP nor HTTP address this issue.

Miscellaneous Protocols

Both FTP and HTTP are TCP based protocols. Two other non-proprietary point-to-point file transfer protocols are used on the internet—both utilise UDP as their transport mechanism.

The Trivial File Transfer Protocol is a lightweight protocol mostly used by network devices to download firmware and configuration parameters. TFTP does not address any security concerns. The File Sharing Protocol[5] (FSP) is a related, but more robust protocol, designed for Internet usage. Since both protocols are UDP based, a large file transfer request is conceptually handled as many small jobs. As such, neither protocol exhibits the busy server behaviour typical of TCP based protocols—instead, a server under heavy load treats all clients equally badly.

Both TFTP and FSP require the client to acknowledge every received UDP packet before sending the next packet. As such, the protocols are very sensitive to latency and do not perform as well as HTTP and FTP. Consequently they are not the protocols of choice for file transfers on the Internet.

Two New Protocols

Two new protocols have been developed that include policies to address the concerns outlined above. Both new protocols are designed to be implemented on top of the existing HTTP protocol.

In describing the behaviour of each protocol, we assume that a large group of independent clients arrive at a server over time to request one or more files. A client knocked back by the server will persist until all its requested transfers have been successfully completed.

‘Reserve’ Protocol

Clients with slow connection speeds obviously take longer to complete their request than faster clients making similar sized requests. With uniformly distributed client connection speeds, at any point in time, the server is likely to be dealing with more slow clients than fast clients. When the server has allocated all its sockets to slow jobs, arriving jobs will be blocked. Scheduling theory dictates that it is sub-optimal for a short job to be held up by a long job. However in both FTP and HTTP, short jobs

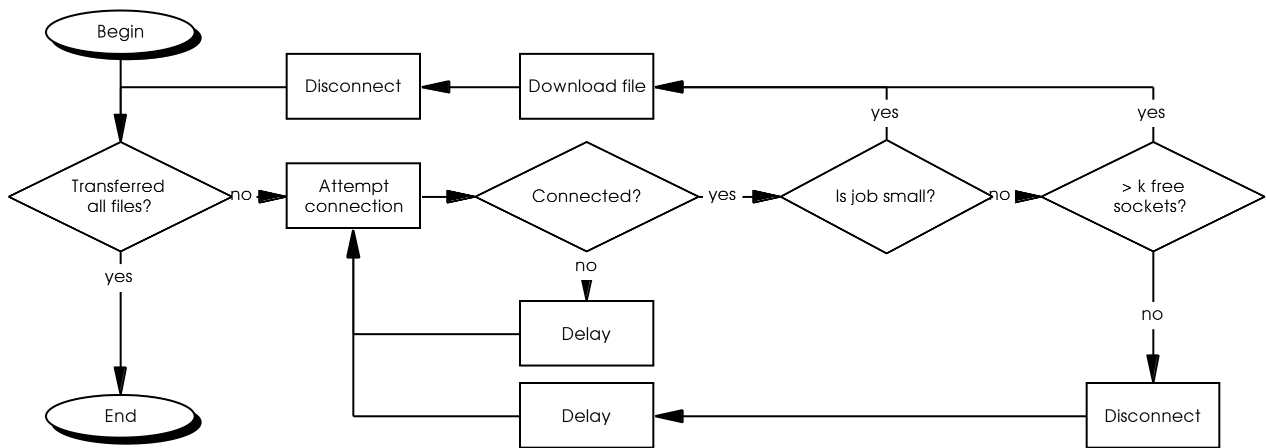

figure 1. **'Reserve' protocol from a client perspective.** The 'Reserve' protocol implements short-job-first by keeping a number of sockets free for short jobs.

may have to wait for long slow jobs. The 'Reserve' protocol implements policy to encourage higher priority for shorter jobs.

In addition, when a server is processing only slow clients it is possible that the aggregate bandwidth demand falls well below the server's capacity. A fully occupied server may be operating at very low efficiency.

To address these issues, we modify HTTP such that the server reserves a proportion of its sockets for small jobs—a maximum of max_large_connections out of n total server sockets may transfer large files at any give time. The remainder of the sockets are reserved for small file transfers. If a client requests a large file when the server is already committed to max_large_connections, the client is immediately disconnected and will retry after some delay. Small requests are handled as per usual. Since the reserved n – max_large_connections sockets have a high turnover, the expected wait on small jobs should be reduced.

By reserving sockets for small jobs, it is rare for all sockets on the server to be busy at any time. Consequently, the number of server-busy rejections is reduced. Thus, if a client requests a resource that is non-existent or unavailable, it can be notified much sooner than under the FTP and HTTP protocols.

'Ticket' Protocol

The 'Reserve' protocol reduces the time taken for short jobs to be admitted to the server. Long jobs will still be turned away from a busy server, and as per FTP and HTTP, 'Reserve' offers no policy to control the admission ordering of rejected jobs. To the server, a rejected job returning has no special rights over a newly arrived job. Thus the best strategy for a rejected client is to repeatedly try for a

connection until a server socket has freed up—the more often a connection is attempted the greater the likelihood of success. This strategy is not the best for the system—numerous clients repeatedly hitting a heavily loaded server creates extra bandwidth demand and reduces the server's ability to deal with currently connected clients.

To address these issues, 'Ticket' builds upon 'Reserve' with the addition of tickets, a wait list and a ready list. A ticket is a sequentially numbered identifier issued by the server to the client that can be presented at a later date as proof of position in a queue. Tickets are issued in response to large job requests, and are initially placed on a wait list. As the size of the ready list drops below a threshold, the lowest numbered wait list tickets are moved to the ready list. The server will only begin a large job when a ticket on the ready list is presented and, like 'Reserve,' only when the number of large jobs in progress is no greater than max_large_connections.

Like 'Reserve,' the 'Ticket' protocol reserves a small percentage of server sockets exclusively for small jobs. 'Ticket' therefore encourages first-come-first-served on large jobs by controlling large job ordering, and encourages shortest-job-first using the policy from 'Reserve.'

The server can use the position of a ticket in the wait list and statistics on recent performance to estimate the time a client should wait before trying again. Unlike the other policies, a rejected client gains no advantage by repeated and rapid attempts to connect to the server for as long as the job is on

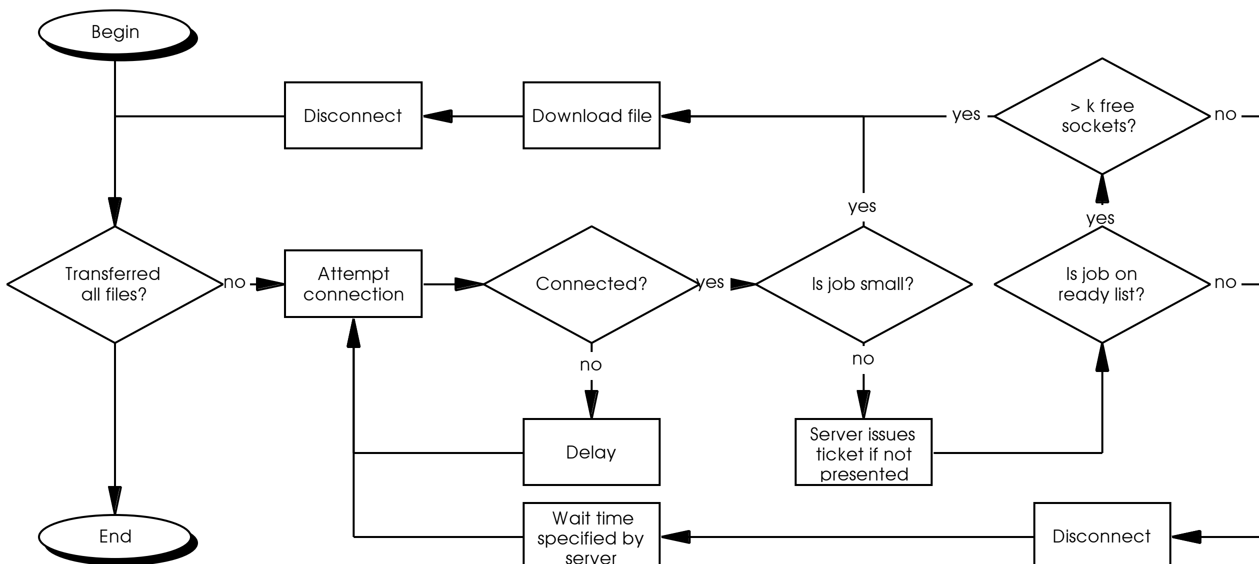

figure 2. **'Ticket' protocol from a client perspective.** Like 'Reserve,' the 'Ticket' protocol implements short-job-first (by keeping a number of sockets free for short jobs) as well as first-come-first-served for long jobs. When a long job arrives at the server, a ticket is issued and placed on the wait list. Only when the server moves the ticket to the ready list can the download begin.

the wait list. The policy does not discourage such behaviour once a client's ticket has reached the ready list.

Note that there is no benefit to the client in requesting multiple tickets for the same file; the first ticket for a resource will appear on the ready list before any subsequently issued tickets. If a client does request multiple tickets and only uses one of them, the unused tickets will eventually move into the ready list and block clients that should otherwise be served. The server can also expect that some clients will not return after being refused entry. A possible solution is give each ticket a time-to-live; if a ticket has been on the wait list for more than the time-to-live interval without being claimed by its owner client, the ticket is deemed unclaimed and will be removed from the ready list, making room for a new ticket from the wait list. Over time the average proportion of unclaimed tickets can be empirically determined and the default ready list size adjusted accordingly.

Under this policy it is possible for tickets to be admitted out of order, however only a fraction of the pending large jobs may be started at any time, so lower-numbered tickets enjoy a higher probability of connection than higher ones. This policy ensures that large jobs are dealt with on a near first-come-first-served basis—thereby evenly distributing waiting times between clients and promoting fairness.

If the server has a number of download sockets free for an extended period of time while there are tickets in the wait list, it can slowly increase the size of the ready list. As the number of large download sockets in use approaches max_large_connections the size of the wait list can be reduced back to the default.

A 'real world' implementation of this policy needs to address the issue of ticket forgery—the policy is useless if clients can make up tickets themselves. A possible solution is for the server to issue a random number with each ticket, which would be stored on the server alongside the ticket. Any ticket not submitted with its assigned random number is to be rejected as counterfeit.

Comparison Protocols

A purpose-built network simulator will model the behaviour of both 'Reserve' and 'Ticket' protocols. To gauge the performance of these new protocols, they will be compared against simulations of the popular HTTP and FTP protocols. For convenience, the models will be labelled 'HTTP' and 'FTP' respectively.

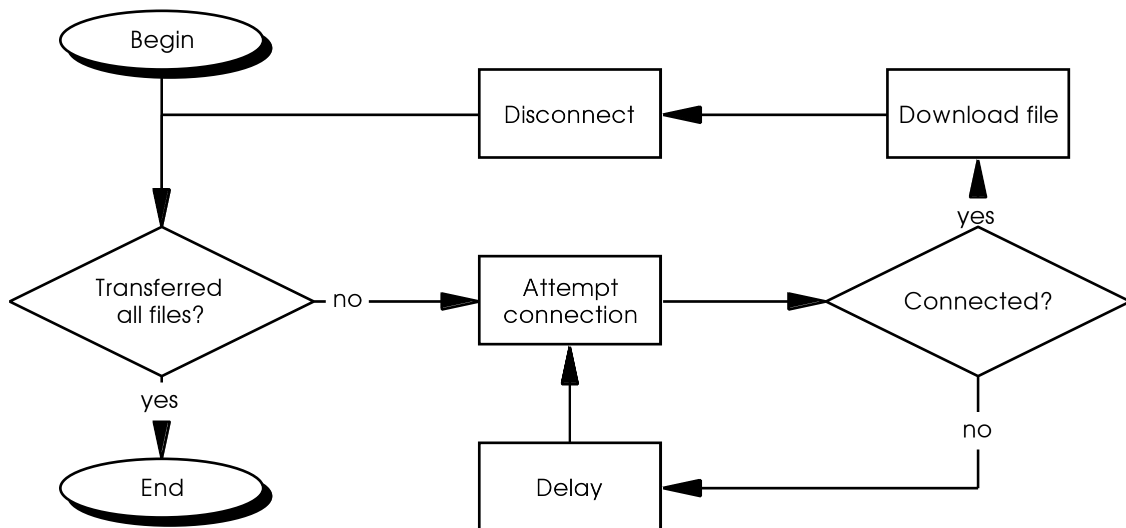

figure 3. **'HTTP' protocol (as modelled) from a client perspective.** HTTP, like 'Ticket' and 'Reserve,' requires the client to make a connection once for each requested file.

'HTTP' Protocol

'HTTP' is based on the HTTP V1.0 protocol. For each requested entity, the client establishes a connection with the server and requests a single file. After the file is transferred the connection is terminated. The client must therefore establish a connection for each file it requests.

Note that HTTP V1.1 allows 'persistent connections' where the client may transfer multiple files during a single connection, should the server allow it. This protocol then behaves much like the FTP protocol—see below.

'FTP' Protocol

'FTP' is based on the FTP protocol. The client establishes a connection with the server and, after some handshaking, the client requests files one at a time. The connection is terminated only after the client has received all its requested files.

The real FTP protocol requires a separate TCP connection for each file to be transferred. In our model the server does not use the general pool of sockets for the data connections. The dual connection mechanism is an artefact of the original design and is not important for the purposes of studying policy.

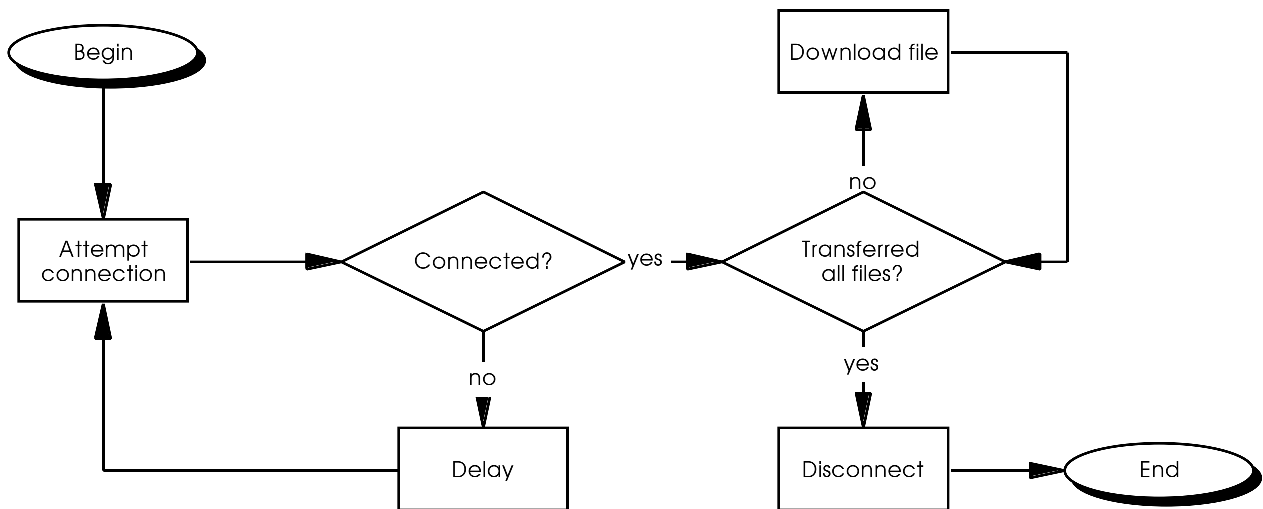

figure 4. **'FTP' protocol (as modelled) from a client perspective.** Note that in a single connection the client can download multiple files.

Simulator Implementation

The simulator models the behaviour of a server and a pool of clients under each of the above protocols and differing load conditions.

Each client 'arrives' at the server at a random offset into the testing period and attempts to download a random number of different-size files from the server. By controlling the random number generator seed it is possible to test each policy using identical client request patterns. Statistics measuring many aspects of the simulation are kept and comparisons can be made between policies.

The simulator, written in Java, uses a thread for each of the server sockets and clients it simulates. Because the simulation is run in real time as a process on a host system, two simulations with identical parameters have been observed to differ by small amounts (less than 5% for total waited time in 5 tests with identical parameters). This small variation in results should be taken into consideration when drawing comparisons.

Experiment 1

Structure

The simulator takes many parameters. While it is important that these values match a real world scenario, it is important to note that each policy is tested under the same set of parameters, so results can be compared knowing that all else remains equal.

The simulator generated 200 clients with arrival times randomly distributed over a 10 minute interval. Figure 5 shows the distribution of clients in our tests.

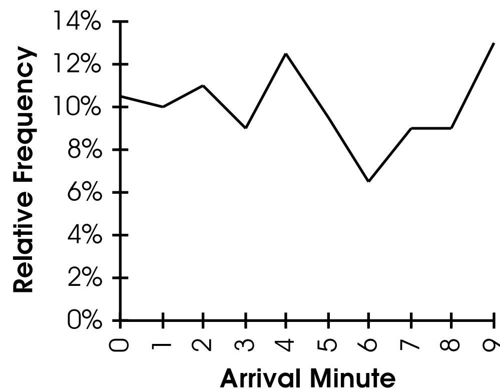

figure 5. **Client arrival time distribution.** 200 clients arrive randomly distributed over a 10 minute interval.

Each client has a randomly chosen connection speed to the server in the range 1600 bytes per second (the approximate throughput of a 14.4 Kbps modem connection) to 35000 bytes per second (approximately the maximum Internet throughput observed in my domain). Clients download one to four files randomly chosen from four available files of sizes 1K, 10K, 100K and 1MB. Accordingly, the frequency distribution of files in Experiment 1 is uniform.

Under simulation, a client connection attempt takes 400ms. If the connection is made, the server takes an additional 100ms to bind the socket to a server socket and begin the session. If the server is too busy to handle a connection (i.e. all available sockets are busy), the client waits 15 seconds before trying again. Both 'Reserve' and 'Ticket' allow the server to tell the client to come back later and try again—in this case the client waits for 30 seconds before trying again.

The server has 30 sockets for the transfer protocol—this number was deliberately chosen to be small so that observable port congestion behaviour occurs. A default server data transfer speed was calculated to be the required speed such that the total server throughput could be transferred within the 10 minute testing window. (Some clients may be limited by their transfer speeds or late start times to transferring only part of their files within in the time window—this was taken into account when calculating the total required server throughput.)

Number of	+0%			+10%				+20%			
Server Sockets	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Transfer	30	30	30	33	33	30	30	36	36	30	30
Reserved	0	0	0	0	0	3	3	0	0	6	6
Total	30	30	30	33	33	33	33	36	36	36	36

figure 6. **Allocation of server sockets across test scenarios.** Each protocol was tested with 30, 33 and 36 server sockets, with the exception of 'Reserve,' which needs extra sockets to keep free. 'Reserve' was only run with 33 and 36 sockets.

The system can be tested under different stresses by modifying the server transfer speeds to more or less than the required values. Each protocol was simulated at 100%, 75% and 50% of required server bandwidth. These loads are referred to as 'sufficient load,' 'light overload,' and 'heavy overload.'

The 'Reserve' and 'Ticket' protocols both require a number of server sockets for small files. Small files are arbitrarily defined to be files less than 20 KB—at most client connection speeds, the time taken to transfer a file less than this threshold is small enough to justify an immediate download. By reserving server sockets, the 'Reserve' and 'Ticket' protocols reduce the number of server sockets that can be serving large files at any point in time. In preliminary tests it was observed that the number of sockets on the server significantly impacted the user experience—having too few sockets exacerbates port contention problems; having too many creates an unserviceable demand on the server.

To allow analysis of the effects of socket allocation the protocols are tested with three different groupings: the default number of 30 server sockets ('+0%'), the default plus 10% ('+10%'), the default plus 20% ('+20%')—the extra sockets in the two latter groupings are reserved for small files when running 'Reserve' or 'Ticket' protocols. The 'Ticket' protocol can be tested without reserved sockets, however the 'Reserve' protocol is meaningless without them. See figure 6 for a summary of the resulting 11 test scenarios:

The 'HTTP' protocol with 30 server sockets is our control, against which we compare the performance of the test clients across the remaining scenarios. In each test the number of clients, their arrival times and there requests are identical; each client in a testing scenario has a corresponding client in the control test.

Two performance measures are used for comparisons.

Relative Average Session Time Improvement

The first measure compares the average client in the testing scenario with the average client in the control test.

A client's session time is the time elapsing between the client's initial arrival at the server through to the completion of all file transfers requested. The session time is inversely proportional to the overall speed of the transfer. The Relative Average Session Time Improvement is the relative time improvement (shorter is improved) of the average of all client session times for a given protocol when compared against the 'HTTP' 30 socket control.

Net Proportion of Clients Faster Than Control

The second measure compares each client in the testing scenario with its corresponding client in the control test, expressing the result as the net number of 'improved' clients.

Specifically, each test client's session time is compared with that of its client counterpart in the control. Because small differences are not relevant to the client, only significant differences are noted, where a significant difference is defined as an absolute time difference both greater than 5% of the control value and greater than 5000 ms. The total count of significantly faster clients less the total number of significantly slower clients expressed as a percentage of total clients is the Net Proportion of Clients Faster Than Control.

If a scenario performs better than control on both measures we can confidently state that the protocol in the scenario is an improvement under the test conditions.

Results

At 'sufficient load' (100% of required bandwidth) and with the default number of server sockets (30), both 'FTP' and 'Ticket' protocols showed improved performance. The 'FTP' improvement is most likely due to the overhead reduction which the protocol enjoys by submitting all its requests in a single connection. Even without reserving sockets for small files, 'Ticket' improves the client experience. When the number of sockets is increased by 10%, the performance improvement over the control is about equal for all four protocols. However in the case of 20% increase socket count both 'HTTP' and 'FTP' drop a little but maintain improvement, whereas 'Reserve' and 'Ticket' flip to become worse than control.

The significant reduction in 'Reserve' and 'Ticket' protocols with +20% sockets is most unexpected. This may be due to the differences in retry times across the policies. When the server refuses connection to clients under 'HTTP' and FTP, in the simulation clients wait for 15 seconds before

Extra Sockets	+0%			+10%				+20%			
Protocol	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Av. Client Time (sec)	102	82	84	82	82	82	84	85	90	123	138
Time Improvement	n/a	19%	17%	20%	20%	20%	18%	17%	12%	-21%	-36%
Slower Clients	n/a	9	14	6	3	5	11	6	7	122	144
Faster Client	n/a	133	123	135	135	134	130	126	101	19	12
Net Faster Clients	n/a	124	109	129	132	129	119	120	94	-103	-132
Net Faster Proportion	n/a	31%	27%	32%	33%	32%	30%	30%	24%	-26%	-33%

figure 7. **Experiment 1, testing under 'sufficient load.'** The chart shows the session time improvement over control (ie speed increase) and the net number of clients that performed better than control. The control is HTTP with 30 sockets under 'sufficient load.'

Extra Sockets	+0%			+10%				+20%			
Protocol	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Av. Client Time (sec)	199	177	168	183	182	201	170	179	186	203	171
Time Improvement	n/a	11%	16%	8%	8%	-1%	15%	10%	6%	-2%	14%
Slower Clients	n/a	46	48	57	62	93	49	59	72	99	52
Faster Client	n/a	83	103	60	73	57	94	88	63	58	95
Net Faster Clients	n/a	37	55	3	11	-36	45	29	-9	-41	43
Net Faster Proportion	n/a	9%	14%	1%	3%	-9%	11%	7%	-2%	-10%	11%

figure 8. **Experiment 1, testing under 'light overload.'** The chart shows the session time improvement over control (ie speed increase) and the net number of clients that performed better than control. The control is HTTP with 30 sockets under 'light overload.'

Extra Sockets	+0%			+10%				+20%			
Protocol	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Av. Client Time (sec)	428	404	386	424	403	413	393	416	416	376	422
Time Improvement	n/a	6%	10%	1%	6%	3%	8%	3%	3%	12%	1%
Slower Clients	n/a	71	67	76	83	71	71	85	95	60	95
Faster Client	n/a	83	87	69	82	84	79	77	76	87	67
Net Faster Clients	n/a	12	20	-7	-1	13	8	-8	-19	27	-28
Net Faster Proportion	n/a	3%	5%	-2%	0%	3%	2%	-2%	-5%	7%	-7%

figure 9. **Experiment 1, testing under 'heavy overload.'** The chart shows the session time improvement over control (ie speed increase) and the net number of clients that performed better than control. The control is HTTP with 30 sockets under 'heavy overload.'

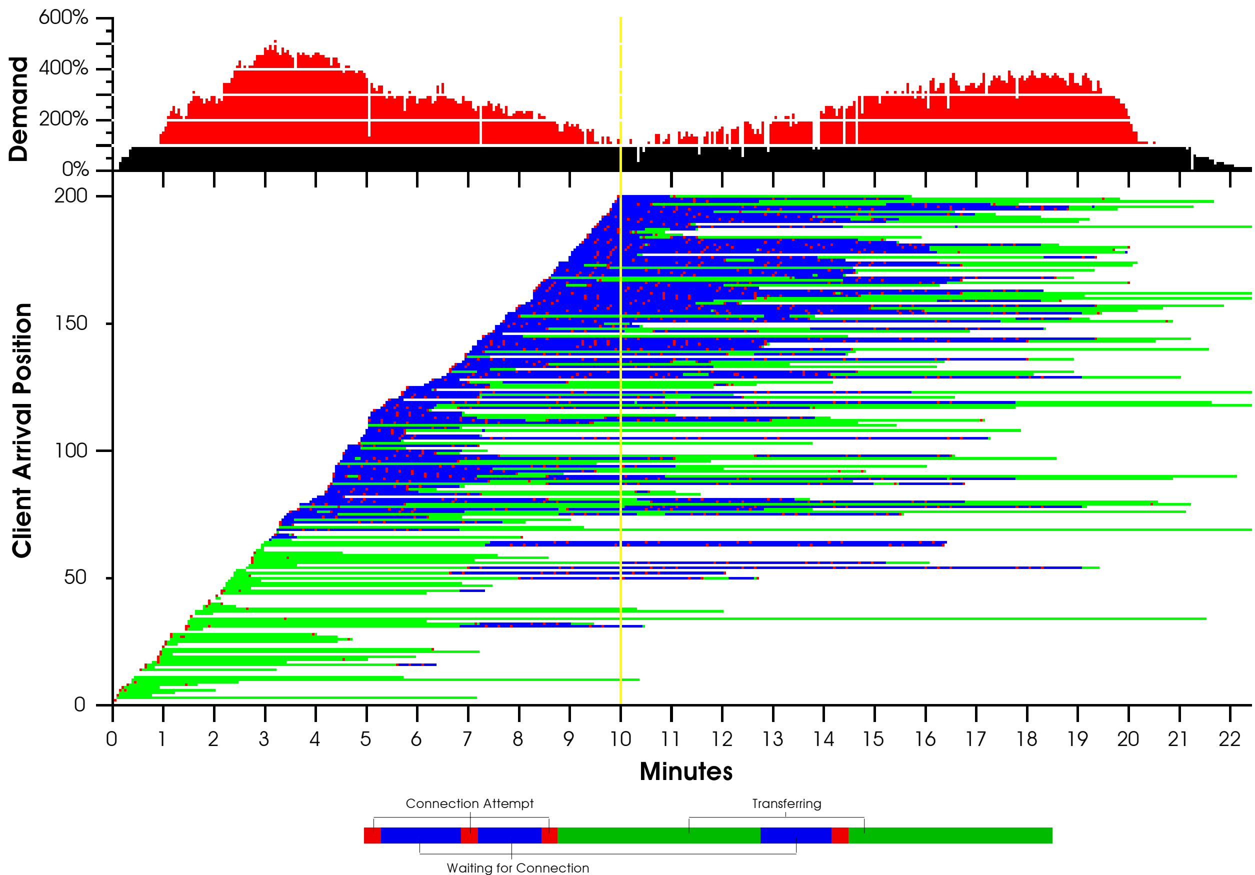

Figure 10. **Experiment 1—'HTTP' Protocol Chronology** in heavy overload condition. 200 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand.

attempting to reconnect, however under the new protocols when the client has a significant number of reserved ports, clients will rarely be refused connection to the server. If the server requests that the client comes back later it will wait a minimum of 30 seconds before attempting to reconnect. The newer protocols appear to be asking the clients to wait for longer than might be necessary, giving the existing protocols a better average client session time.

Under light overload most of the scenarios show a small performance increase over the control. The 'Reserve' scenarios do not obey this trend—in both cases there is a slight, but not significant worsening of the average client time, and a small net excess of degraded clients. In the 'FTP' scenarios with increased sockets, the number of clients improved is cancelled out by the number of clients degraded.

As the load on the server increases, the relative performance differences have reduced to the point where, on average, the protocols show relative performance improvements near 0%.

The results above, as graphically summarised in figures 14 and 15, show that increasing the number of sockets on the server does not generally lead to any significant performance increase for both

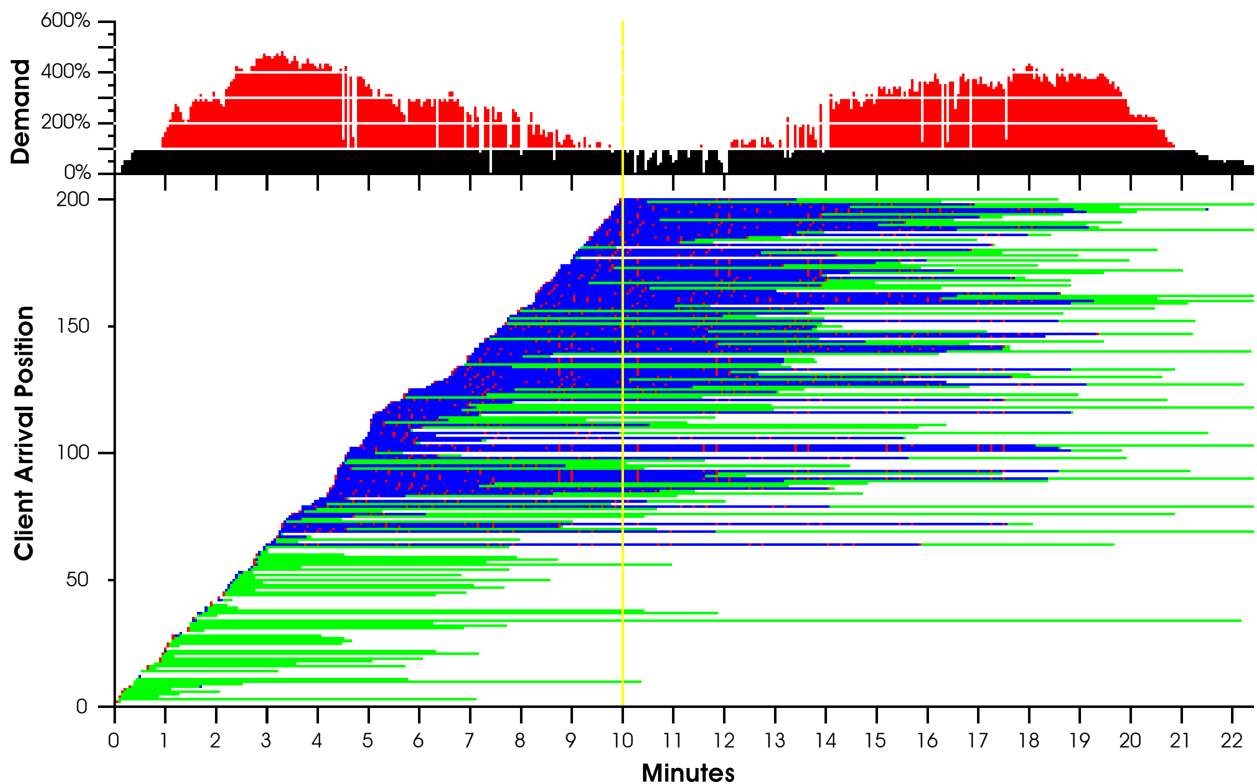

Figure 11. **Experiment 1—'FTP' Protocol Chronology** in heavy overload condition. 200 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand.

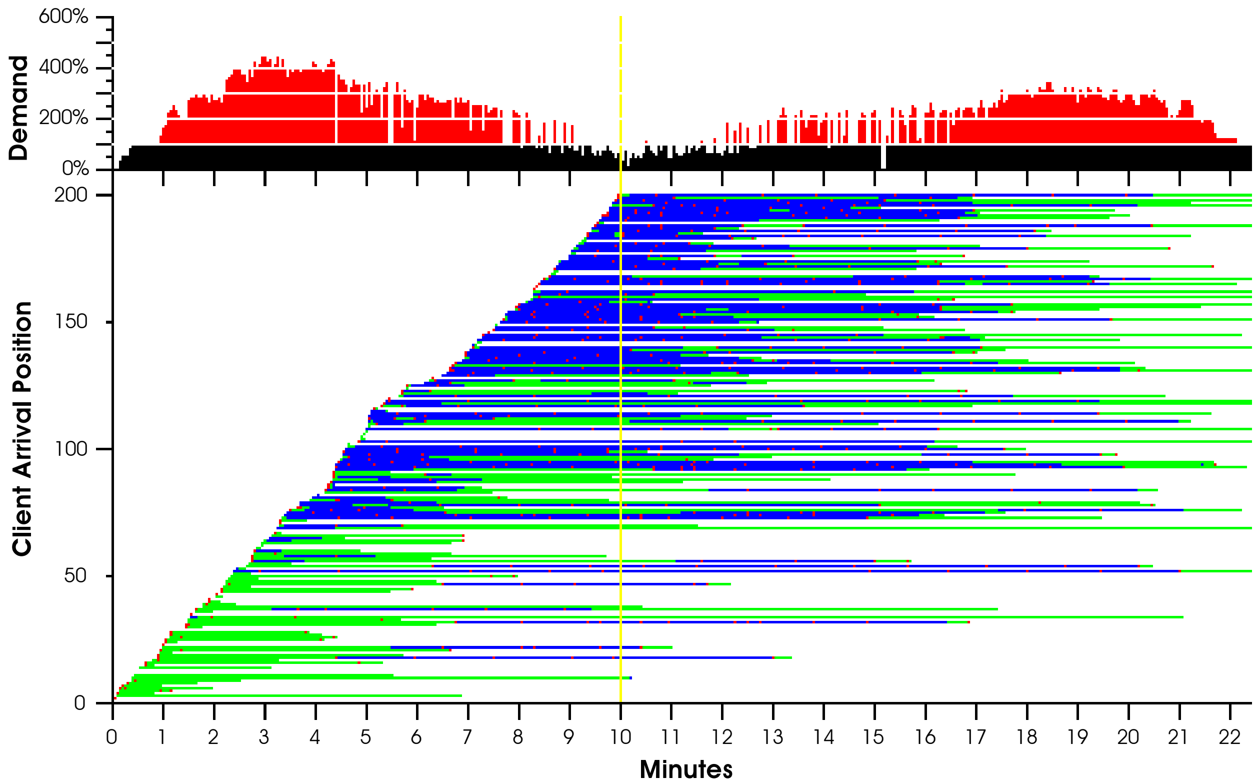

Figure 12. **Experiment 1—'Reserve' Protocol Chronology** in heavy overload condition. 200 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand.

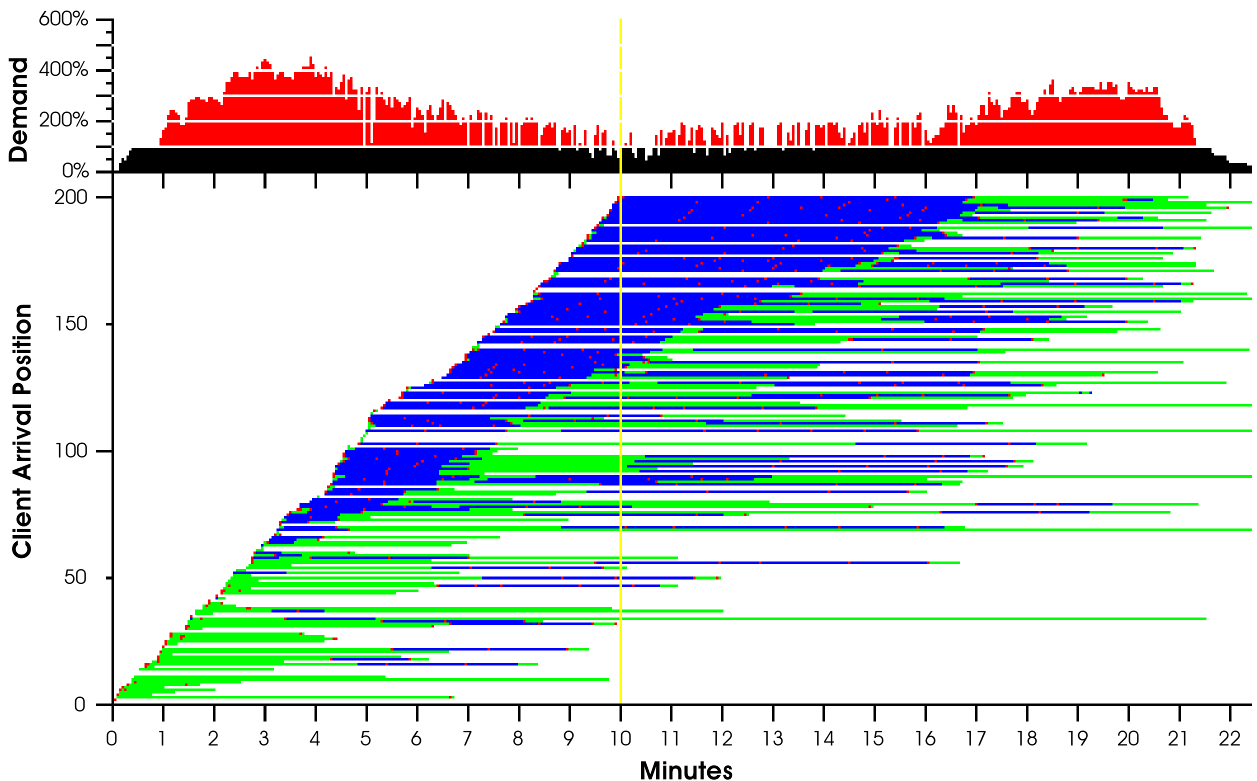

Figure 13. **Experiment 1—'Ticket' Protocol Chronology** in heavy overload condition. 200 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand.

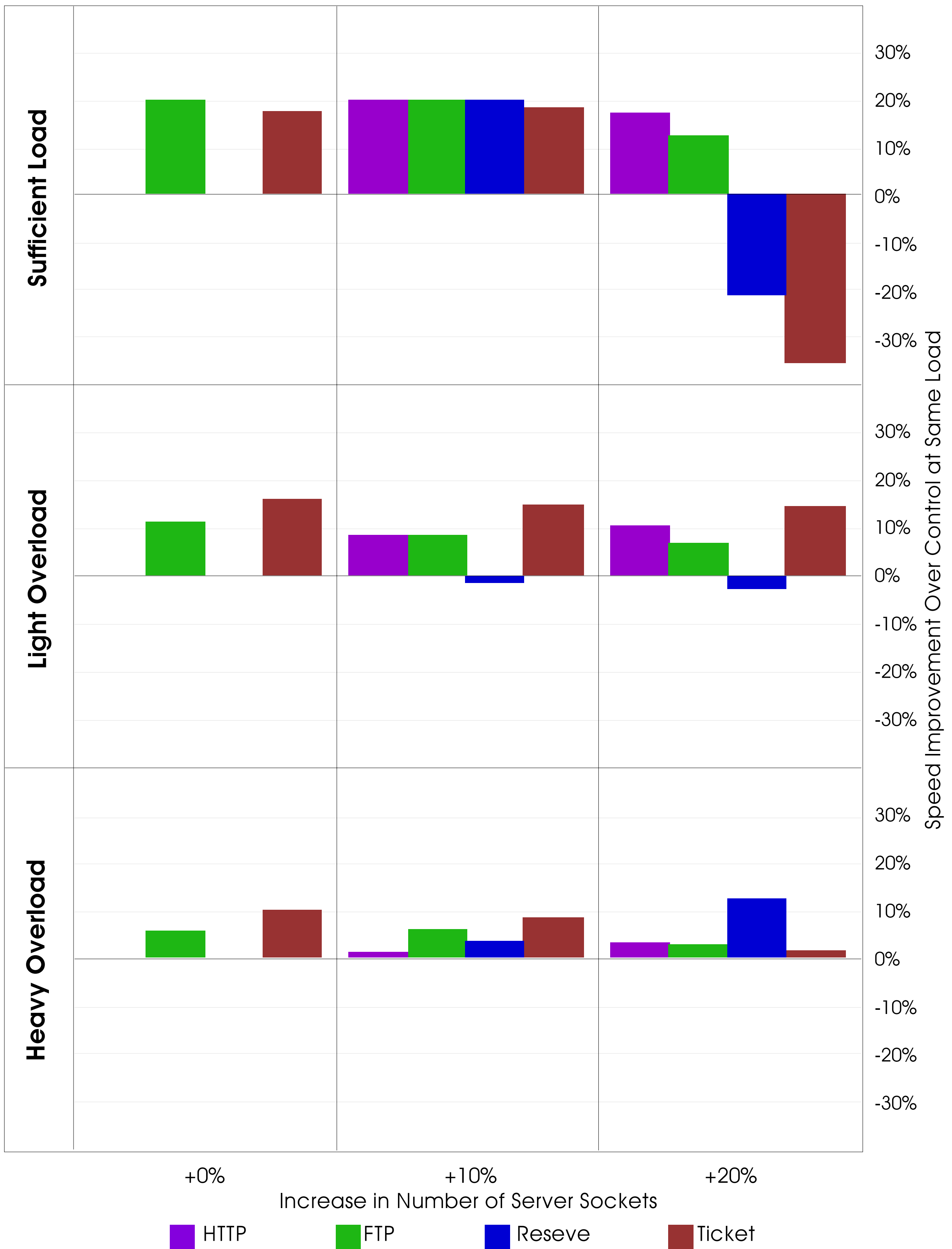

Figure 14. **Experiment 1—Relative protocol speed improvements across all tests.** For each of the three server loadings 11 tests were run—one control test (‘HTTP’ with 30 sockets) and 10 protocol tests. Each bar represents the relative speed improvement over the control in that particular server loading category. Note that there is no bar for ‘Reserve’ at 30 sockets—when there are no extra sockets to reserve the protocol is undefined.

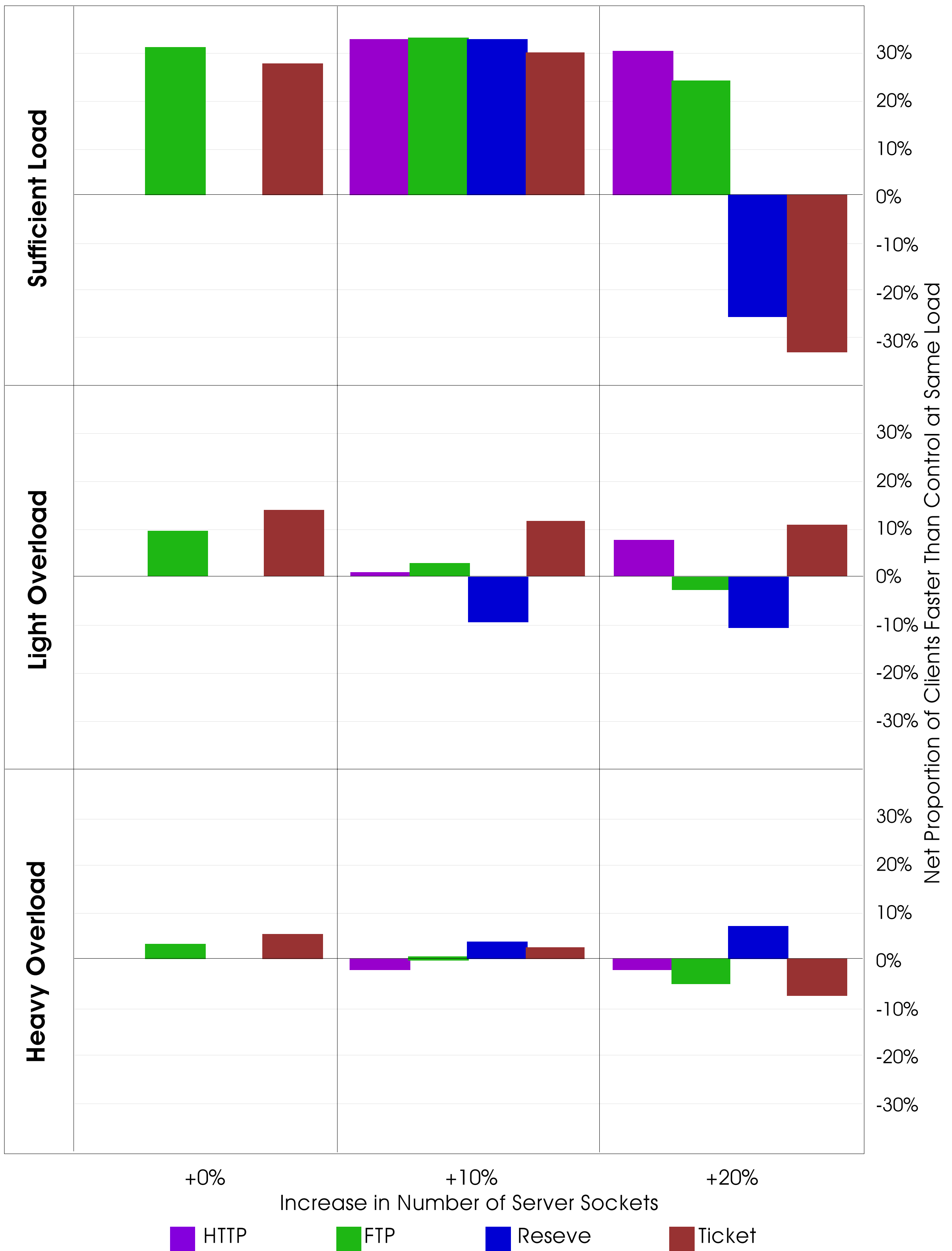

Figure 15. **Experiment 1—Net proportion of clients performing better than control across all tests.** For each of the three server loadings 11 tests were run—one control test ('HTTP' with 30 sockets) and 10 protocol tests. Each bar represents the net proportion of clients whose speed significantly improved relative to the corresponding client in the control in that particular server loading category. Note that there is no bar for 'Reserve' at 30 sockets—when there are no extra sockets to reserve the protocol is undefined.

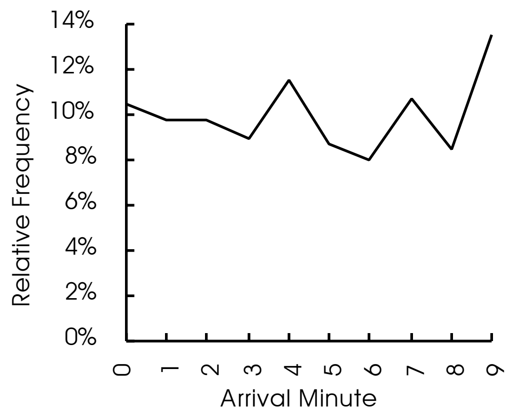

figure 16. **Client arrival time distribution**. 400 clients arrive randomly distributed over a 10 minute interval.

'HTTP' and 'FTP' protocols. The most significant finding is that although the 'Reserve' and 'Ticket' protocols show small improvement under some conditions, they never perform significantly better, but sometimes significantly worse than the 'HTTP' and 'FTP' protocols.

Under these test conditions the policies introduced in the new protocols appear to be having little positive impact on server performance. The existing file transfer policies are rarely worse and sometimes better than the proposed protocols. A possible explanation for the disappointing results to this point is the small number of clients with very small jobs. In the above scenarios only 9% of clients request a total of under 10KB, only 20% request less than the 20KB immediate download threshold. 60% of all clients requested the largest file. The server does not hit the targeted load for 2 to 3 minutes on average—so relatively few of the total 200 jobs are short-job clients which might benefit from the proposed protocols.

A typical Internet server can expect to be dealing with a higher proportion of short-job clients, so it is not unreasonable to increase the proportion of short-job clients arriving at the simulated server.

Experiment 2

Structure

In Experiment 2 we address the above concerns by increasing the proportion of short-job clients. Under the new settings, clients request between one and three files from a selection of seven files—four are 1KB, two are 10KB and one is 1MB. As a result, the distribution is skewed towards smaller jobs; 34% of clients request under 10KB, 65% request under the 20KB threshold and only 29% request the largest file.

These are realistic figures—clients connecting to web servers are likely to pull down a few small HTML files and several GIF/JPEG image files before they decide to request a large file. Data from Arlitt and Williamson[6] indicates that over 80% of the requests from a survey of large web sites are for files less than 100K, and approximately 10% of file requests are for files larger than 1MB.

By increasing the proportion of small jobs, we should see more clients directly benefiting from the short-job-first policy. In Experiment 2 the number of clients connecting has been increased to 400 so we can better observe the effects of the policies on port congestion.

As per Experiment 1, clients' arrival times are randomly distributed over a 10 minute interval—see figure 16. Aside from the changes in number of clients and composition of client jobs, all other parameters are the same as for Experiment 1.

Results

Under sufficient load our tests indicated that 'FTP' benefited from an increase in the number of sockets. The 'Reserve' protocol showed a slight improvement, and 'Ticket' showed a larger, but underwhelming improvement.

Extra Sockets	+0%			+10%				+20%			
Protocol	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Av. Client Time (sec)	119	113	110	125	97	112	103	107	97	112	103
Time Improvement	n/a	5%	7%	-5%	18%	5%	13%	10%	18%	5%	13%
Slower Clients	n/a	66	51	105	26	72	34	52	40	72	47
Faster Client	n/a	96	94	70	121	94	102	101	123	94	107
Net Faster Clients	n/a	30	43	-35	95	22	68	49	83	22	60
Net Faster Proportion	n/a	8%	11%	-9%	24%	6%	17%	12%	21%	6%	15%

figure 17. **Experiment 2, testing under 'sufficient load.'** The chart shows the session time improvement over control (ie speed increase) and the net number of clients that performed better than control. The control is HTTP with 30 sockets under 'sufficient load.'

Extra Sockets	+0%			+10%				+20%			
Protocol	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Av. Client Time (sec)	332	427	186	323	333	203	187	441	459	183	182
Time Improvement	n/a	-29%	44%	3%	0%	39%	44%	-33%	-38%	45%	45%
Slower Clients	n/a	185	49	148	159	72	60	194	220	56	50
Faster Client	n/a	96	194	118	127	184	196	75	83	197	203
Net Faster Clients	n/a	-89	145	-30	-32	112	136	-119	-137	141	153
Net Faster Proportion	n/a	-22%	36%	-8%	-8%	28%	34%	-30%	-34%	35%	38%

figure 18. **Experiment 2, testing under 'light overload.'** The chart shows the session time improvement over control (ie speed increase) and the net number of clients that performed better than control. The control is HTTP with 30 sockets under 'light overload.'

When the server operates under light overload, 'FTP' performs worse than the 'HTTP' control scenario largely due to its behaviour of tying up server sockets with slow-job clients. Increasing the number of sockets slightly reduces the congestion and increases performance to the same level as the control. When the number of sockets is upped by 20%, both 'HTTP' and 'FTP' take a significant performance hit. This is probably because the each client now receives a smaller cut of the available bandwidth.

The new protocols begin to show promise—average client time is reduced by 30-40% and the net number of clients with improved performance represents at least 28% of all clients. Under the conditions of this experiment, 'HTTP' and 'FTP' protocols block a significant number of clients. 'Reserve' and 'Ticket' protocols appear to reduce the blockage by ensuring that short jobs are processed at highest priority.

At heavy overload with the standard 30 sockets, 'FTP' performance is very poor—73% slower than the control. However it appears that 'FTP' is very sensitive to the number of server sockets—with 10% extra sockets it only performs 15% slower than control. When given a further 10% extra sockets 'FTP' performs 43% better than control; by adding just 6 more sockets the average client session times for 'FTP' were reduced by 67%. Such a difference was unexpected, but can be attributed to the sensitive interplay between two variables: when given more sockets the waiting time is reduced (since

Extra Sockets	+0%			+10%				+20%			
Protocol	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Av. Client Time (sec)	1265	2184	375	1434	1455	506	375	1380	717	355	353
Time Improvement	n/a	-73%	70%	-13%	-15%	60%	70%	-9%	43%	72%	72%
Slower Clients	n/a	257	18	190	201	51	26	168	95	24	29
Faster Client	n/a	57	289	104	110	259	282	123	235	285	287
Net Faster Clients	n/a	-200	271	-86	-91	208	256	-45	140	261	258
Net Faster Proportion	n/a	-50%	68%	-22%	-23%	52%	64%	-11%	35%	65%	65%

figure 19. **Experiment 2, testing under 'heavy overload.'** The chart shows the session time improvement over control (ie speed increase) and the net number of clients that performed better than control. The control is HTTP with 30 sockets under 'heavy overload.'

	Sockets	Ideal	+0%			+10%				20%			
	Policy	Rel. Time	HTTP	FTP	Ticket	HTTP	FTP	Reserve	Ticket	HTTP	FTP	Reserve	Ticket
Load	Sufficient	100%	100%	95%	93%	105%	82%	95%	87%	90%	82%	95%	87%
	Light Over	133%	280%	360%	157%	272%	281%	172%	158%	372%	387%	154%	153%
	Heavy Over	200%	1066%	1842%	316%	1209%	1227%	427%	316%	1164%	604%	300%	297%

figure 20. **Relative client session times.** A comparison of the performance of the protocols across the different server loading factors.

the server is busy less often) but the transfer time is increased (since the limited bandwidth is distributed between more active clients). These two effects compete against each other—the relationship of these variables warrants further research.

Interestingly, the performance of 'HTTP' is largely unaffected by the addition of extra server sockets. Since clients don't stay connected as long as for 'FTP,' the port congestion is significantly reduced.

The 'Reserve' and 'Ticket' protocols perform very well under heavy load. The new protocols consistently outperform traditional protocols in both measures by large margins. In the 36 socket case, the average client session time for 'Ticket' is 25% of the 'HTTP' protocol clients. Client session times are being significantly reduced by the application of our scheduling policies.

Further insight on relative performance is gained by comparing the performance of the protocols across the different loads.

In the heavy overload condition the server has half the bandwidth required to satisfy all clients, so, ideally, client session times should double. We know from queuing theory that this is far too optimistic—our data shows that the session time is considerably more sensitive to crowding at the server. Under HTTP, doubling server loads increases client session times by over 1000%. As noted, 'FTP' can benefit from increased socket count—this is reflected in our data.

'Reserve' and 'Ticket' perform nearly as well in the heavy overload condition as 'HTTP' and 'FTP' do in the light overload condition. Both new protocols, with extra sockets, are much closer to the ideal time increases than the established protocols. Hence, the new protocols are considerably less sensitive to overloading.

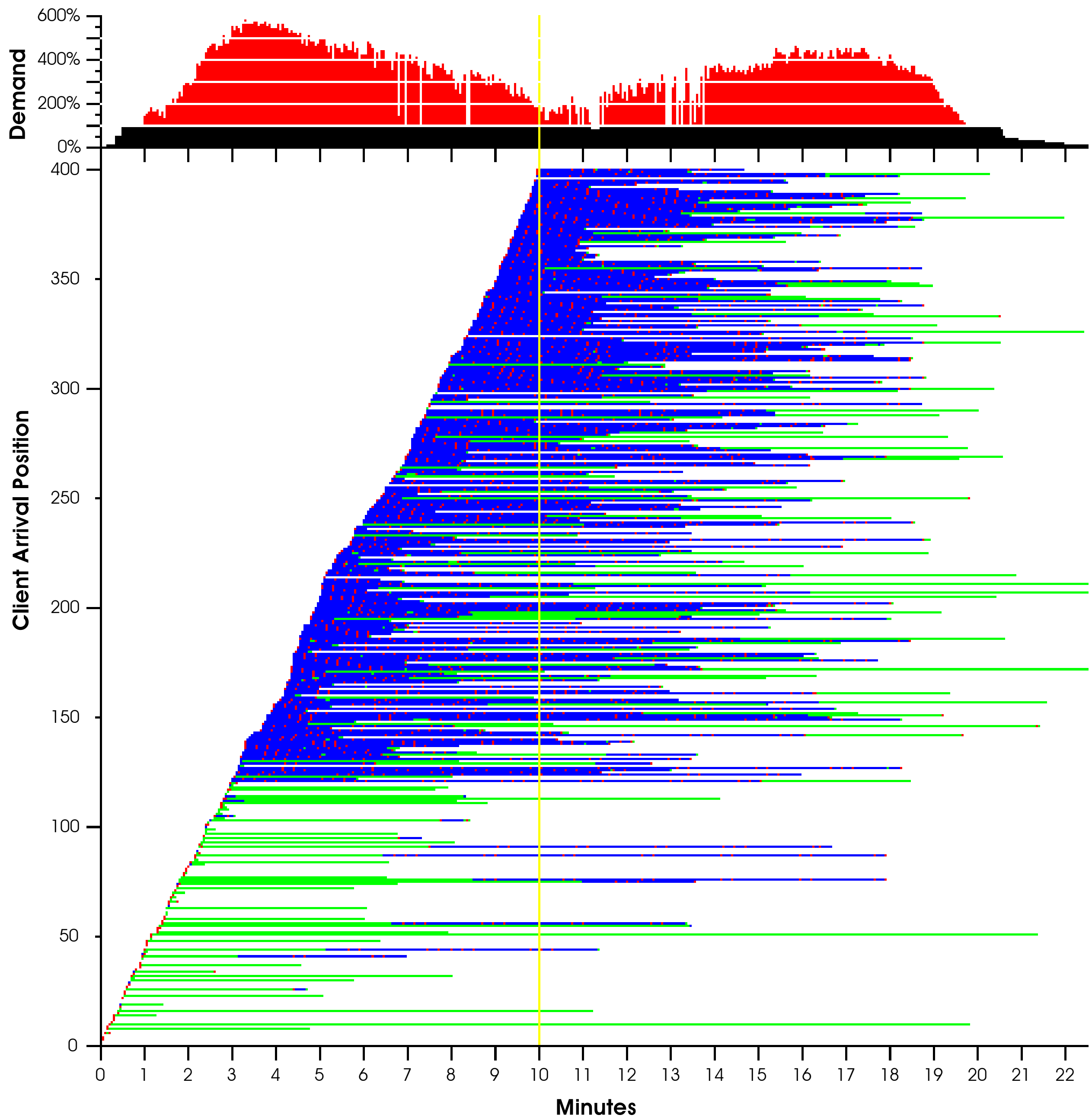

Figure 21. **Experiment 2—'HTTP' Protocol Chronology** in heavy overload condition. 400 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand.

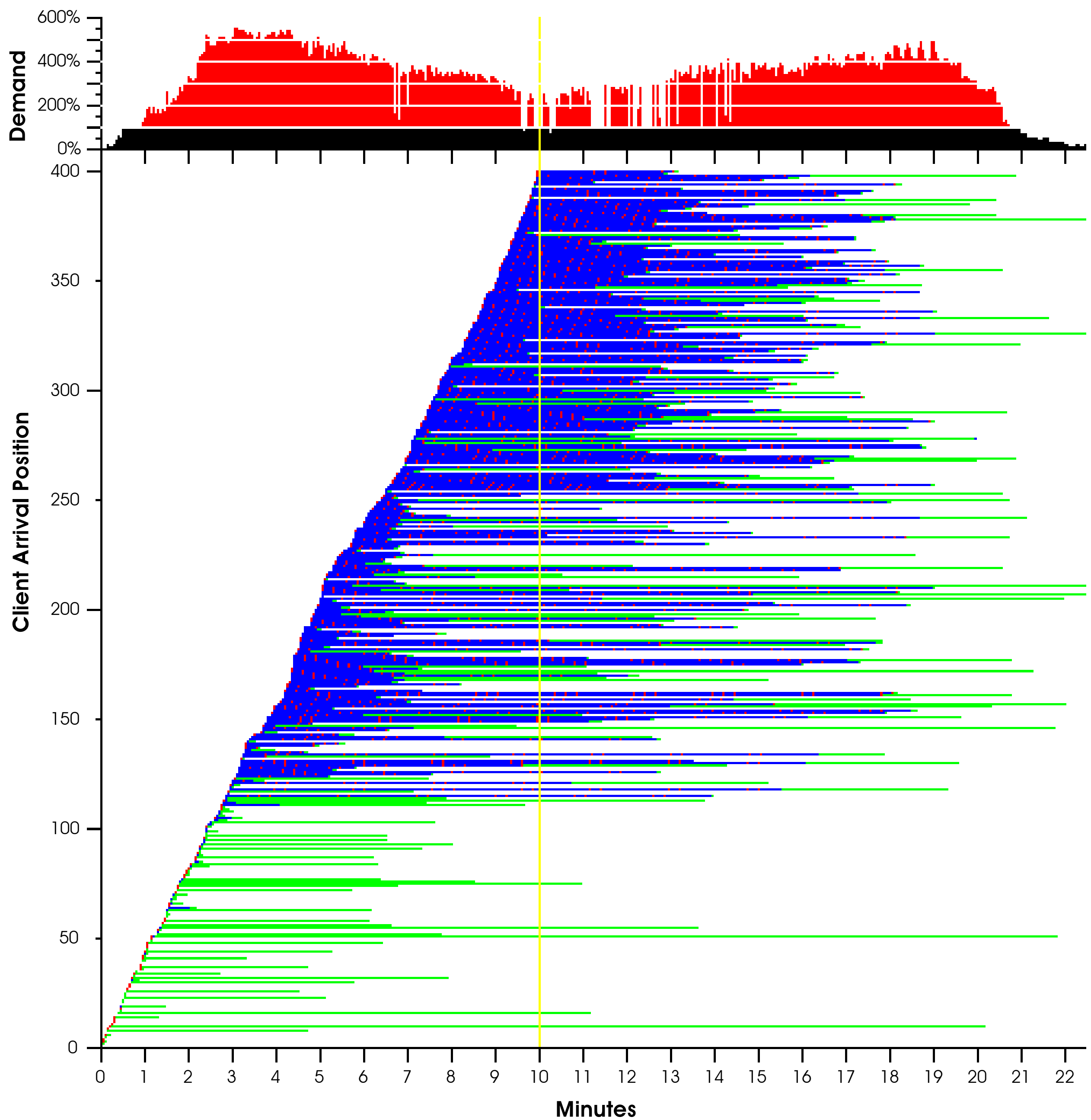

Figure 22. **Experiment 2—'FTP' Protocol Chronology** in heavy overload condition. 400 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand.

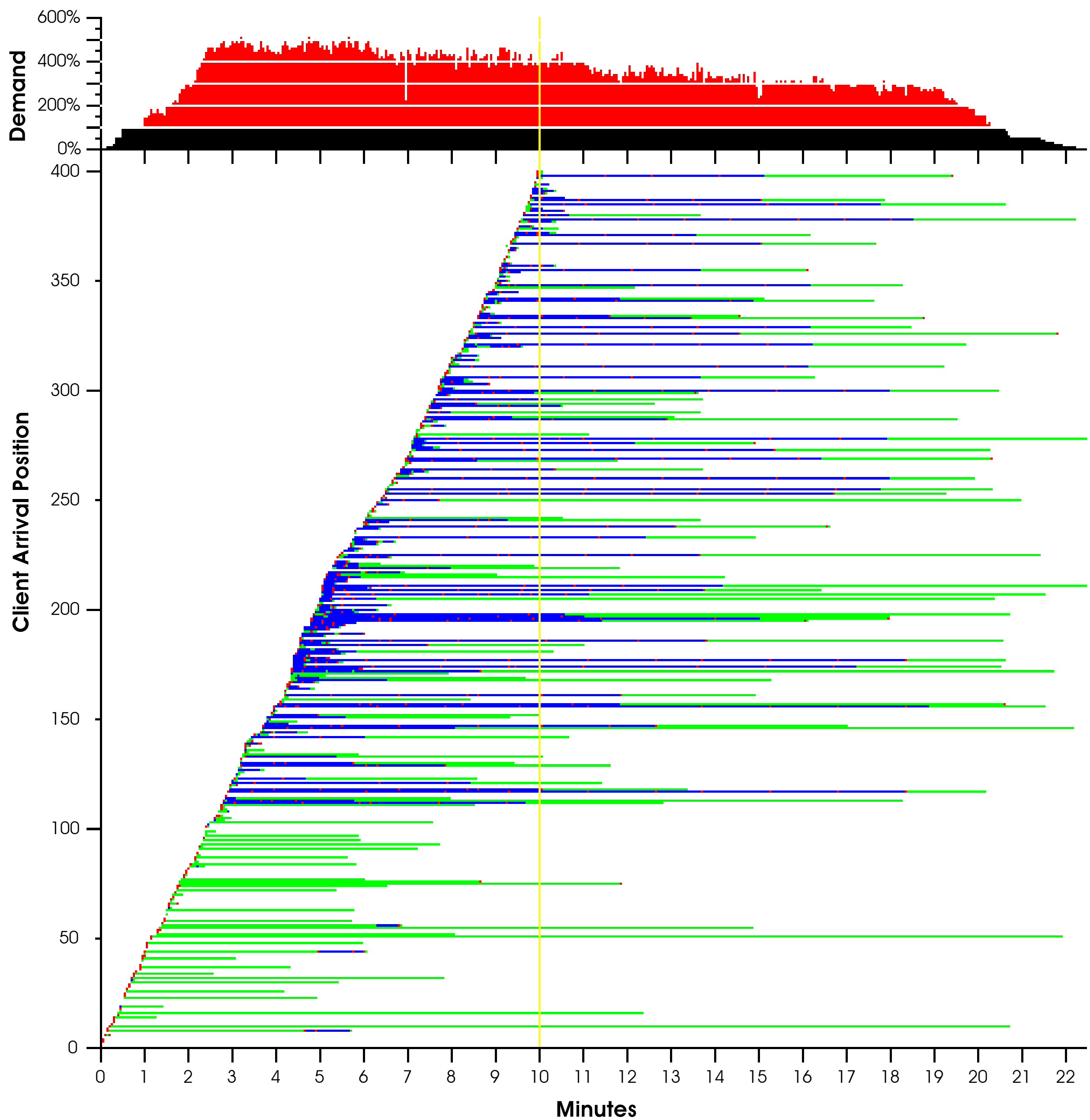

Figure 23. **Experiment 2—'Reserve' Protocol Chronology** in heavy overload condition. 400 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand. Notice the dramatic reduction in blue area—indicating significantly reduced waiting times.

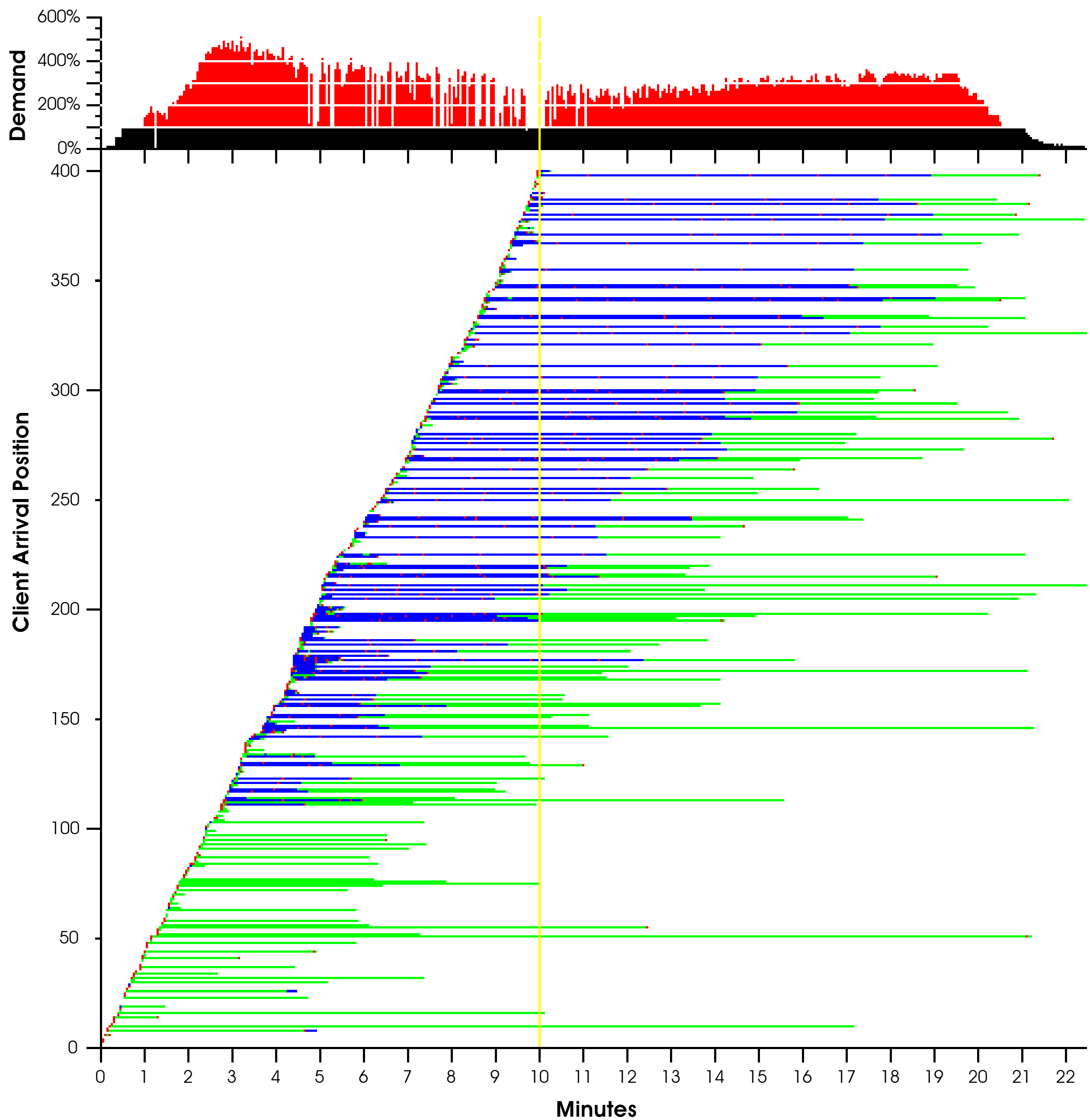

Figure 24. **Experiment 2—'Ticket' Protocol Chronology** in heavy overload condition. 400 clients attempt to connect over 10 minutes to a 36 socket server. Horizontal bars indicate client status: the beginning of the bar marks the beginning of a client's session; green sections signify connected clients; blue sections signify active but not connected; red dots represent connection attempts. The upper portion of the graph shows the demand on the server; black portion shows demand satisfied by the server; red indicates excess demand. Again we see waiting times are much lower than 'HTTP' and 'FTP'—also note that the blue/green interface is close to linear. Clients on this line are being admitted on a first-come-first-served basis.

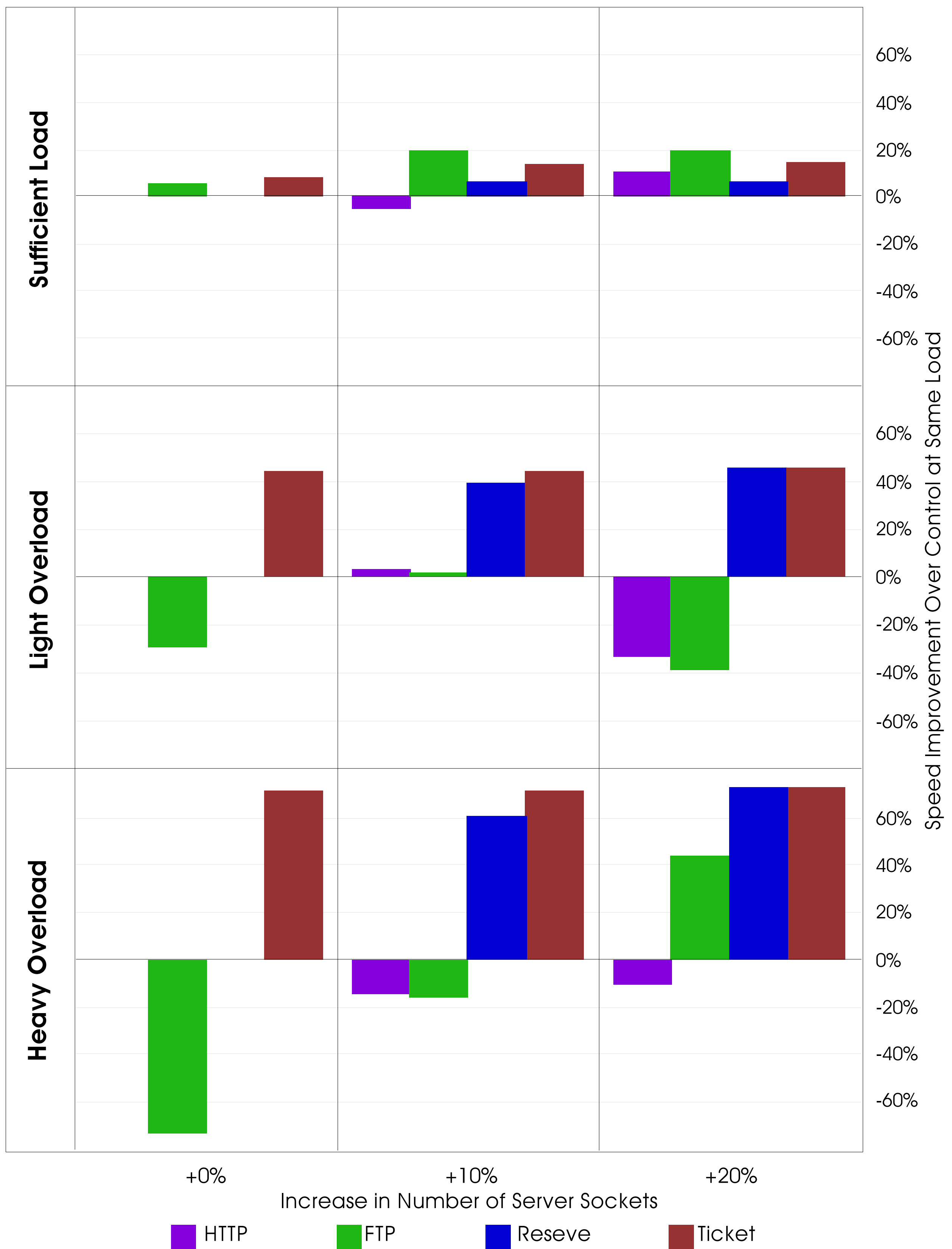

Figure 25. **Experiment 2—Relative protocol speed improvements across all tests.** For each of the three server loadings 11 tests were run—one control test ('HTTP' with 30 sockets) and 10 protocol tests. Each bar represents the relative speed improvement over the control in that particular server loading category. Note that there is no bar for 'Reserve' at 30 sockets—when there are no extra sockets to reserve the protocol is undefined.

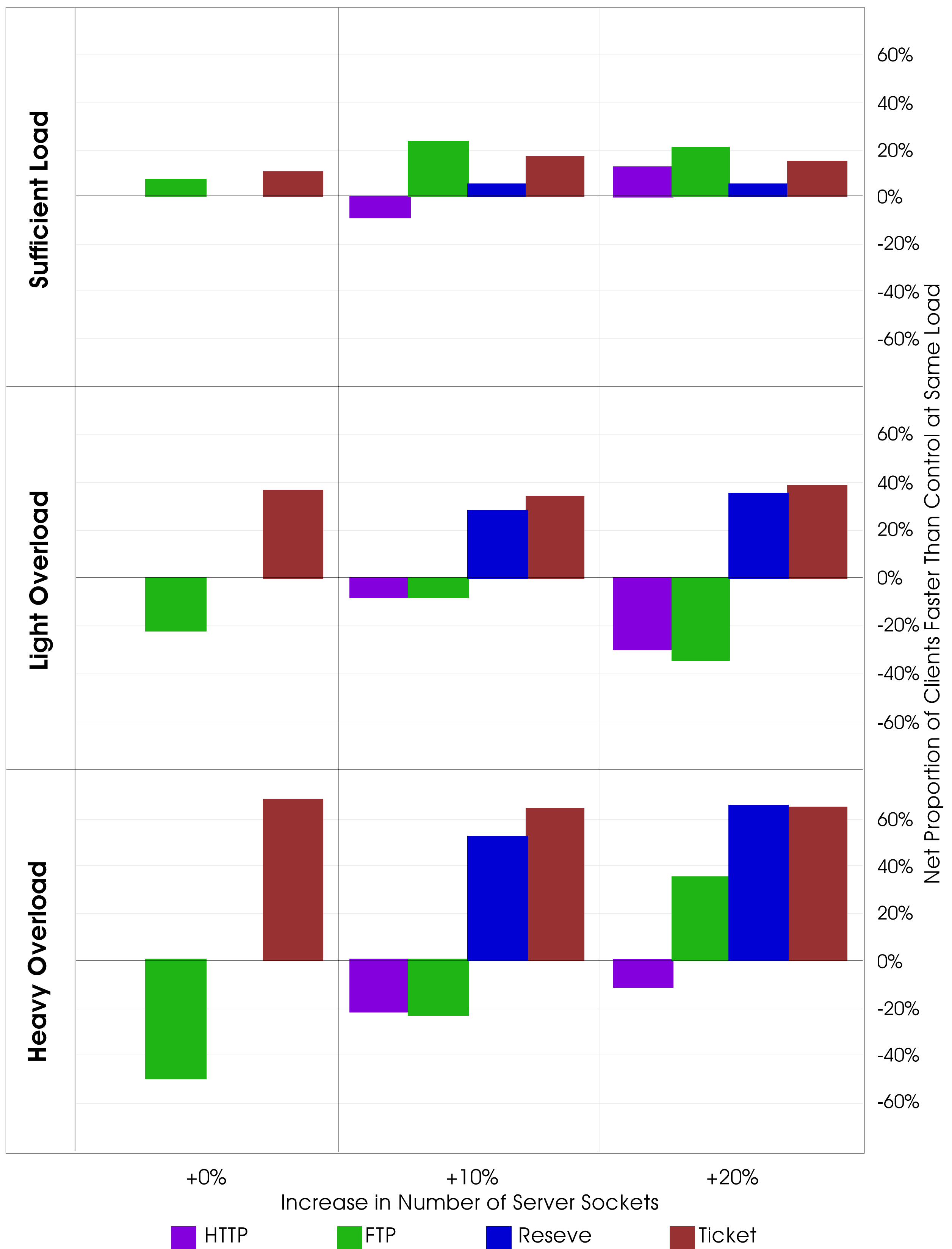

Figure 26. **Experiment 2—Net proportion of clients performing better than control across all tests.** For each of the three server loadings 11 tests were run—one control test ('HTTP' with 30 sockets) and 10 protocol tests. Each bar represents the net proportion of clients whose speed significantly improved relative to the corresponding client in the control in that particular server loading category. Note that there is no bar for 'Reserve' at 30 sockets—when there are no extra sockets to reserve the protocol is undefined.

Discussion

The two policies taken from processor scheduling, shortest-job-first and first-come-first-served, were successfully adapted to the network versions short-jobs-first and large-job-first-come-first-served. Two new protocols, 'Reserve' and 'Ticket,' were built from these policies.

In the case of overloaded servers with a mix of many short jobs and some much longer jobs, the simulator showed that the two new protocols offered a great improvement in client experience over the current protocols. Running the new protocols, the server degrades performance more gracefully than HTTP and FTP. Under sufficient load conditions, the new protocols offered little improvement. When the mix of jobs included relatively few short jobs and server load was low, the two protocols performed worse than the HTTP and FTP modelled protocols.

The new policies should only be implemented in situations of high load and a mix of relatively few long jobs. Both policies were added to an HTTP base, and can be switched in and out. It is feasible to develop a server that only begins short-job-first as the ratio of short-jobs to long jobs increases pass a threshold, and ticketing as the server becomes overloaded.

The 'Ticket' protocol is basically the 'Reserve' protocol with the addition of the first-come-first-served policy. In processor scheduling this policy does not improve average job times, so it is no surprise that 'Ticket' rarely performs much better than 'Reserve.' Although it is more work to implement the ticking policies for little speed gain, I believe it has some key properties that make it worthy for use. Under the 'Reserve' protocol, it is still in a client's best interest to rapidly poll a busy server, adding unnecessary bandwidth demands to the server and network—under 'Ticket' this strategy is of no advantage to a wait listed client, and such 'bad behaviour' can be punished.

Additionally, ticketing opens up interesting commercial possibilities on the Internet. A company selling software can issue tickets to buyers. Only on presenting the ticket can the user download the software. Alternatively, preferred-status customers could submit a customer identification token with a ticket to ensure that their ticket moves through the wait list at an accelerated rate. The ticket policy is largely implemented at the server—all a client needs to do is present tickets it has been given and defer reconnection until a server specified time. As such, all fine tuning and sub-policies relating to tickets can be made at the server without requiring changes to the client software.

The current protocols are not able to supply information such as the average client wait time, or the number of times a client must persist to get a connection—ticketing gives us access to these statistics.

The provision of useful statistics, reduction in client connection busy-waiting, and opportunities for better customer service make the 'Ticket' protocol model particularly attractive.

In summary, I have shown that client experience on the Internet may be improved by extending HTTP to include two scheduling policies: short-job-first and large-job-first-come-first-served.

Footnotes

1 The FTP protocol is fully described in RFC 959, see <http://andrew2.andrew.cmu.edu/rfc/rfc959.html>

2 See <ftp://ietf.org/ietf/ftpext/ftpext-charter.txt> for the charter of an IETF working group that hopes to address some of these problems.

3 HTTP v1.0 is fully described in RFC 1945, see <http://andrew2.andrew.cmu.edu/rfc/rfc1945.html>

4 See RFC 2068 at <http://andrew2.andrew.cmu.edu/rfc/rfc2068.html>

5 For more information on FSP, see <http://www.softlab.ece.ntua.gr/faq/fsp_faq.html>.

6 Arlitt, M and Williamson, C *Web Server Workload Charachterization: The Search For Invariants* 1996 ACM SIGMETRICS Conference Proceedings, Philadelphia, PA May 1996.

Bibliography

Postel J. and Reynolds, J. *File Transfer Protocol (FTP)* 1985, <http://andrew2.andrew.cmu.edu/rfc/rfc959.html>

T. Berners-Lee, T., Fielding R., and Frystyk, H. *Hypertext Transfer Protocol—HTTP/1.0* 1996, <http://andrew2.andrew.cmu.edu/rfc/rfc1945.html>

Fielding, R., Gettys, J., Mogul J., Frystyk, H, Berners-Lee T. *Hypertext Transfer Protocol—HTTP/1.1* 1997, <http://andrew2.andrew.cmu.edu/rfc/rfc2068.html>

Doherty, A. F*ile Service Protocol (FSP)—Frequently Asked Questions* 1996, <http://www.softlab.ece.ntua.gr/faq/fsp_faq.html>

Arlitt, M and Williamson, C *Web Server Workload Charachterization: The Search For Invariants* 1996 ACM SIGMETRICS Conference Proceedings, Philadelphia, PA May 1996. also at <ftp://ftp.cs.usask.ca/pub/discus/paper.96-3.ps.Z>