The invention provides improved data or other information flow control over a distributed computing or information storage/retrieval network. The flow, movement, or migration of information is controlled to minimize the data transfer latency and to prevent overloads. A first or outgoing flow control block and procedure controls the outgoing flow of data (both requests and response) on the network connection and makes sure that no data is sent before the previous portions of data are received by a network peer in order to minimize the connection latency. A second or Q-algorithm block and procedure controls the stream of the requests arriving on the connection and decides which of them should be broadcast to the neighbors. Its goal is to make sure that the responses to these requests would not overload the outgoing bandwidth of this connection. A third or fairness block makes sure that the connection is not monopolized by any of the logical request/response streams from the other connections. It allows to multiplex the logical streams on the connection, making sure that every stream has its own fair share of the connection bandwidth regardless of how much data are the other streams capable of sending. These blocks and the functionality they provide may be used separately or in conjunction with each other. As the inventive method, procedures, and algorithms may advantageously be implemented as computer programs, such as computer programs in the form of software, firmware, or the like, the invention also advantageously provides a computer program and computer program product when stored on tangible media. Such computer programs may be executed on appropriate computer or information appliances as are known in the art, and may typically include a processor and memory coupled to the processor.
FLOW CONTROL METHOD FOR DISTRIBUTED
BROADCAST-ROUTE NETWORKS

Related Applications

This application claims the benefit of United States Provisional Application No. 60/281,324 filed April 3, 2001 and entitled "Flow Control Method for Distributed Broadcast-Route Networks," which is incorporated herein by reference in its entirety. This application is related to United States Patent Application No. 09/724,937 filed November 28, 2000 and entitled "System, Method and Computer Program for Flow Control In a Distributed Broadcast-Route Network With Reliable Transport Links;" herein incorporated by reference and enclosed as Appendix D.

Field of Invention

This invention pertains generally to systems and methods for communicating information over an interconnected network of information appliances or computers, more particularly to system and method for controlling the flow of information over a distributed information network having broadcast-route network and reliable transport link network characteristics, and most particularly to particular procedures, algorithms, and computer programs for facilitating and/or optimizing the flow of information over such networks.

BACKGROUND

The Gnutella network does not have a central server and consists of the number of equal-rights hosts, each of which can act in both the client and the server capacity. These hosts are called ‘servents’. Every servent is connected to at least one other servent, although the typical number of connections (links) should be more than two (the default number is four). The resulting network is highly redundant with many possible ways to go from one host to another. The connections (links) are the reliable TCP connections.

When the servent wishes to find something on the network, it issues a request with a globally unique 128-bit identifier (ID) on all its connections, asking the neighbors to send a response if they have a requested piece of data (file) relevant to the request. Regardless of whether the servent receiving the request has the file or not, it propagates (broadcasts) the request on all other links it has, and remembers that any responses to the request with this ID should be sent back on the link which the request has arrived from. After that if the request with the same ID arrives on the other link, it is dropped and no action is taken by the receiving servent in order to avoid the ‘request looping’ which would cause an excessive network load.

Thus ideally the request is propagated throughout the whole Gnutella network (GNnet), eventually reaching every servent then currently connected to the network. The forward propagation of the requests is called ‘broadcasting’, and the sending of the responses back is called ‘routing’. Sometimes both broadcasting and routing are referred to as the ‘routing’ capacity of the servent, as opposed to its client (issuing the request.
and downloading the file) and server (answering the request and file-serving) functions. In a Gnutella network each node or workstation acts as a client and as a server.

Unfortunately the propagation of the request throughout the whole network might be difficult to achieve in practice. Every servent is also the client, so from time to time it issues its own requests. Thus if the propagation of the requests is unlimited, it is easy to see that as more and more servents join the GNnet, at some point the total number of requests being routed through an average servent will overload the capacity of the servent physical link to the network.

Since the TCP link used by the Gnutella servents is reliable, this condition manifests itself by the connection refusal to accept more data, by the increased latency (data transfer delay) on the connection, or by both of these at once. At that point the Gnutella servent can do one of three things: (i) it can drop the connection, (ii) it can drop the data (request or response), or (iii) it can try to buffer the data in hope that it will be able to send it later.

The precise action to undertake is not specified, so the different implementations choose different ways to deal with that condition, but it does not matter – all three methods result in serious problems for the GNnet, namely one of A, B, or C, as follows: (A) Dropping the connection causes the links to go up and down all the time, so many requests and responses are simply lost, because by the time the servent has to route the response back, the connection to route it to is no longer available. (B) Dropping the data (request or response) can lead to a response being dropped, which overloads the network by unnecessarily broadcasting the requests over hundreds of servents only to drop the responses later. (C) Buffering the data increases the latency even more. And since it does little or nothing to fix the basic underlying problem (an attempt to transmit more data than the network is physically capable of) it only causes the servents to eventually run out of memory. To avoid that, they have to resort to other two ways of dealing with the connection overload albeit with much higher link latency.

These problems were at least somewhat anticipated by the creators of the Gnutella protocol, so the protocol has a built-in means to limit the request propagation through the network, called ‘hop count’ and ‘TTL’ (time to live). Every request starts its lifecycle with a hop count of zero and TTL of some finite value (de facto default is 7). As the servent broadcasts the request, it increases its hop count by one. When the request hop count reaches the TTL value, the request is not broadcast anymore. So the number of hosts N that see the request can be approximately defined by the equation:

\[
 N = (\text{avLinks} - 1)^{\text{TTL}}, \quad (\text{EQ. 1})
\]

where avLinks is the average number of the servent connections, and the TTL is the TTL value of the request. For the avLinks = 5 and TTL = 7 this comes to a value of N of about 10,000 servents.

Unfortunately the TTL value and the number of links are typically hard-coded into the servent software and/or set by the user. In any case, there’s no way for the servent to quickly (or dynamically) react to the
changes in the GNet data flow intensity or the data link capacity. This leads to the state of affairs when the GNet is capable of functioning normally only when the number of servents in the network is relatively small or they are not actively looking for data. When either of these conditions is not fulfilled, the typical servant connections are overloaded with the negative consequences outlined elsewhere in this description. Put simply, the GNet enters the ‘meltdown’ state with the number of ‘visible’ (searchable from the average servant) hosts dropping from the range of between about 1,000-4,000 to a much smaller range or between about100-400 or less, which decreases the amount of searchable data by a factor of ten or about an order of magnitude. At the same time the search delay (the time needed for the request to traverse 7 hops (the default) or so and to return back as a response) climbs to hundreds of seconds. Response time on the order of hundreds of seconds are typically not tolerated by users, or at the very least are found to be highly irritating and objectionable.

In fact, the delay becomes so high that the servant routing tables (the data structures used to determine which connection the response should be routed to) reach the full capacity, overflow and time out even before the response arrives so that no response is ever received by the requestor. This, in turn, narrows the search scope even more, effectively making the Gnutella unusable from the user standpoint, because it cannot fulfill its stated goal of being the file searching tool.

The ‘meltdown’ described above has been observed on the Gnutella network, but in fact the basic underlying problem is deeper and manifests itself even with a relatively small number of hosts, when the GNet is not yet in an actual meltdown state.

The problem is that the GNet uses the reliable TCP protocol or connection as a transport mechanism to exchange messages (requests and responses) between the servents. Being the reliable vehicle, the TCP protocol tries to reliably deliver the data without paying much attention to the delivery latency (link delay). Its main concern is the reliability, so as soon as the data stream exceeds the physical link capacity, the TCP tries to buffer the data itself in a fashion, which is not controlled by the developer or the user. Essentially, the TCP code hopes that this data burst is just a temporary condition and that it will be able to send the buffered data later.

When the GNet is not in a meltdown state, this might even be true – the burst might be a short one. But regardless of the nature of the burst, this buffering increases the delay. For example, when a servant has a 40 kbits/sec modem physical link shared between four connections, every connection is roughly capable of transmitting and receiving about 1 kilobyte of data per second. When the servant tries to transmit more, the TCP won’t tell the servant application that it has a problem until it runs out of TCP buffers, which are typically of about 8 kilobyte size.

So even before the servant realizes that its TCP connections are overloaded and has any chance to remedy the situation, the link delay reaches 8 seconds. Even if just two servants along the 7-hop request/response path are in this state, the search delay exceeds 30 seconds (two 8-second delays in the request path and two – in the response path). Given the fact that the GNet typically consists of the servents with very different communication capabilities, the probability is high that at least some of the servents in the request path will be overloaded. Actually this is exactly what can be observed on the Gnutella network even when it is not in
the meltdown state despite the fact that most of the servants are perfectly capable of routing data with a sub-
second delay and the total search time should not exceed 10 seconds.

Basically, the ‘meltdown’ is just a manifestation of this basic problem as more and more servants
become overloaded and eventually the number of the overloaded servants reaches the ‘critical mass’, effectively
making the GNet unusable from a practical standpoint.

It is important to realize that there’s nothing a servant can do to fight this delay -- it does not even know
that the delay exists as long as the TCP internal buffers are not yet filled to capacity.

Some developers have suggested that UDP be used as the transport protocol to deal with this situation,
however, the proposed attempts to use UDP as a transport protocol instead of TCP are likely to fail. The reason
for this likely failure is that typically the link-level protocol has its own buffers. For example, in case of the
modem link it might be a PPP buffer in the modem software. This buffer can hold as much as 4 seconds of data,
and though it is less than the TCP one (it is shared between all connections sharing the physical link), it still can
result in a 56-second delay over seven request and seven response hops. And this number is still much higher
than the technically possible value of less than ten seconds and, what is more important, higher than the
perceived delay of the competing Web search engines (such as for example AltaVista, Google, and the like), so
it exceeds the user expectations set by the ‘normal’ search methods.

Therefore, there remains a need for a system, method, and computer program and communication
protocol that minimizes the latency and reduces or prevents GNet or other distributed network overload as the
number of servants grows.

There also remains a need for particular methods, procedures, algorithms, and computer programs for
facilitating and optimizing communication over such distributed networks and for allowing such networks to be
scaled over a broad range.

BRIEF DESCRIPTION OF DRAWINGS

Fig. 1. The Gnutella router diagram.
Fig. 2. The Connection block diagram.
Fig. 3. The bandwidth layout with a negligible request volume.
Fig. 4. The bandwidth reservation layout.
Fig. 5. The ‘GNet leaf’ configuration.
Fig. 6. The finite-size request rate averaging.
Fig. 7. Graphical representation of the ‘herringbone stair’ algorithm.
Fig. 8. Hop-layered request buffer layout in the continuous traffic case.
Fig. 9. Request buffer clearing algorithm.
Fig. 10. Hop-layered round-robin algorithm.
Fig. 11. Request buffer Q-volume and data available to the RR-algorithm.
Fig. 12. The response distribution over time (continuous traffic case).
Fig. 13. Equation (62) integration trajectory in (tau, t) space.

Fig. 14. Sample Rt(t)*r(t, tau) peak distribution in (tau, t) space in the discrete traffic case.

Fig. 15. Rt(t)*r(t, tau) value interpolation and integration in the discrete traffic case.

Fig. 16. Rt(t)*r(t, tau) integration tied to the Q-algorithm step size.

Fig. 17. Single response interpolation within two Q-algorithm steps.

SUMMARY

The invention provides improved data or other information flow control over a distributed computing or information storage/retrieval network. The flow, movement, or migration of information is controlled to minimize the data transfer latency and to prevent overloads. A first or outgoing flow control block and procedure controls the outgoing flow of data (both requests and responses) on the network connection and makes sure that no data is sent before the previous portions of data are received by a network peer in order to minimize the connection latency. A second or Q-algorithm block and procedure controls the stream of the requests arriving on the connection and decides which of them should be broadcast to the neighbors. Its goal is to make sure that the responses to these requests would not overload the outgoing bandwidth of this connection. A third or fairness block makes sure that the connection is not monopolized by any of the logical request/response streams from the other connections. It allows to multiplex the logical streams on the connection, making sure that every stream has its own fair share of the connection bandwidth regardless of how much data are the other streams capable of sending. These blocks and the functionality they provide may be used separately or in conjunction with each other. As the inventive method, procedures, and algorithms may advantageously be implemented as computer programs, such as computer programs in the form of software, firmware, or the like, the invention also advantageously provides a computer program and computer program product when stored on tangible media. Such computer programs may be executed on appropriate computer or information appliances as are known in the art, and may typically include a processor and memory couple to the processor.

DETAILED DESCRIPTION OF EMBODIMENTS

Exemplary embodiments of the inventive system, method, algorithms, and procedures are now described relative to the drawings. For the convenience of the reader, the description is organized into sections as outlined below. It will be appreciated that aspects of the invention are described throughout the specification and that the section notations and headers are merely for the convenience of the reader and do not limit the applicability or scope of the description in any way.

1. Introduction
2. Finite message size consequences for the flow control algorithm
3. Gnutella router building blocks
4. Connection block diagram
5. Blocks affected by the finite message size
6. Packet size and sending time
   6.1. Packet size
6.2. Packet sending time
7. Packet layout and bandwidth sharing
  7.1. Simplified bandwidth layout
  7.2. Packet layout
  7.3. ‘Herringbone stair’ algorithm
  7.4. Multi-source ‘herringbone stair’
8. Q-algorithm implementation
  8.1. Q-algorithm latency
  8.2. Response/request ratio and delay
    8.2.1. Instant response/request ratio
    8.2.2. Instant delay value
9. Recapitulation of Selected Embodiments
10. References
    Appendix A. ‘Connection 0’ and request processing block
    Appendix B. Q-algorithm step size and numerical integration
    Appendix C. OFC GUID layout and operation
    Appendix D. U.S. Pat. App. Serial No. 09/724,937 (Reference [1])

1. Introduction.

The inventive algorithm is directed toward achieving the infinite scalability of the distributed networks, which use the ‘broadcast-route’ method to propagate the requests through the network in case of the finite message size. The ‘broadcast-route’ here means the method of the request propagation when the host broadcasts the request it receives on every connection it has except the one it came from and later routes the responses back to that connection. ‘Finite message size’ means that the messages (requests and responses) can have the size comparable to the network packet size and are ‘atomic’ in a sense that another message transfer cannot interrupt the transfer of the message. That is, the first byte of the subsequent message can be sent over the communication channel only after the last byte of the previous message.

Even though the algorithm described below can be used for various networks with the ‘broadcast-route’ architecture, the primary target of the algorithm is the Gnutella network, which is widely used as the distributed file search and exchange system. The system and method may as well be applied to other networks and are not limited to Gnutella networks. The Gnutella protocol specifications (herein incorporated by reference) are known, incorporated by reference herein, and can be found at the web sites identified below, the contents of which are incorporated by reference:

http://gnutella.wego.com/go/wego.pages.page?groupId=116705&view=page&pageId=119598&folderId=116767&panelId=-1&action=view
http://www.gnutelladev.com/docs/capnbry-protocol.html
http://www.gnutelladev.com/docs/our-protocol.html

-6-
To achieve the infinite scalability of the network, it is desirable to have some sort of the flow control algorithm built into it. Such an algorithm for Gnutella and other similar 'broadcast-route' networks was described in United States Patent Application Serial No. 09/724,937 filed 28 November 2000 and entitled System, Method and Computer Program for Flow Control In a Distributed Broadcast-Route Network With Reliable Transport Links, herein incorporated by reference and enclosed as Appendix D, and identified as reference [1] in the remainder of this description. The flow control procedure and algorithm was designed on an assumption that the messages can be broken into the arbitrarily small pieces (continuous traffic case). This is not always the case – for example, the Gnutella messages are atomic in a sense mentioned above (several messages cannot be sent simultaneously over the same link) and can be quite large - several kilobytes. Thus it is desirable to adopt the continuous-traffic flow control algorithm to the situation when the messages are atomic and have finite size (discrete traffic case). This adaptation and the algorithms that achieve it are the subject of this specification. At the same time this document describes some further details of a particular flow control implementation.

2. Finite message size consequences for the flow control algorithm.

The flow control algorithm described in [1] uses the continuous-space equations to monitor and control the traffic flows and loads on the network. That is, all the variables are assumed to be the infinite-precision floating-point numbers. For example, the typical equation ([1], Eq. 13 – describes the rate of the traffic to be passed to other connections) might look like this:

\[ x = (Q - u) / Rav \]

where \( x \) is the rate of the incoming forward-traffic (requests) passed by the Q-algorithm to be broadcast on other connections.

The direct implementation of such equations would mean that when, say, 40 bytes of requests would arrive on the connection, the Q-algorithm might require that 25.3456 bytes of this data should be forwarded for the broadcast and 14.6544 bytes should be dropped. This would not be possible for two reasons – first, it is not possible to send a non-integer number of bytes, and second, these 40 bytes might represent a single request.

The first obstacle is not very serious – after all, we might send 25 bytes and drop 15 bytes. The resulting error would not be a big one, and a good algorithm should be tolerant to the computational and rounding errors of such magnitude.

The second obstacle is worse – since the message (in this case, request) is atomic, it is not possible to break it into two parts, one of which would be sent, and another would be dropped. We have to drop or to send the whole request as an atomic unit. Thus regardless of whether we decide to send or to drop the messages
which cannot be fully sent, the Q-algorithm would treat all the messages in the same way, effectively passing all the incoming messages for broadcast or dropping all of them. Such a behavior would introduce an error, which would be too large to be tolerated by any conceivable flow control algorithm, so it is clearly unacceptable and we have to invent some way to deal with this situation.

The similar problem arises when the fair bandwidth-sharing algorithm tries to allocate the space for the requests and responses in the packet to be sent out. Let’s say we would like to evenly share the 512-byte packet between requests and responses, and it turns out that we have twenty 30-byte requests and a single 300-byte response — what should one do? Should one send a 510-byte packet with the response and 7 requests, and then send a 90-byte packet with 3 responses, or should we send a 600-byte packet with a response and 10 requests? The first decision would not evenly share the packet space and bandwidth, possibly resulting in the unfair bandwidth distribution, and the second would increase the connection latency because of the increased packet size. And what if the response is bigger than 512 bytes to begin with?

Such decisions can have a significant effect on the flow control algorithm behavior and should not be taken lightly. So first of all, let’s draw a diagram of the Gnutella message routing node and see where are the blocks where these decisions will have to be made.

The Fig. 1 presents the high-level block diagram of the Gnutella router (the part of the servant responsible for the message sending and receiving):

![Diagram of Gnutella router](image)

Fig. 1. The Gnutella router diagram.

Essentially the router consists of several TCP connection blocks, each of which handles the incoming and outgoing data streams from and to another servant and of the virtual Connection 0 block. The latter handles the stream of requests and responses of the router’s servant User Interface and of the Request Processing block. This block is called ‘Connection 0’, since the data from it is handled by the flow control algorithms of all other connection in a uniform fashion – as if it has come from the normal TCP Connection block. (See, for example, the description of the fairness block in [1].)

As far as the TCP connections are concerned, the only difference between Connection 0 and any TCP connection is that the requests arriving from this “virtual” connection might have a hop value equal to −1. This would mean that these requests have not arrived from the network, but rather from the servant User Interface Block through the “virtual” connection – these requests have never been transferred through the Gnutella network (GNet). The diagram shows that Connection 0 interacts with the servant UI Block through some API; there are no requirements to this API other than the natural one - that the router and the UI Block developers should be in agreement about it. In fact, this API might closely mimic the normal Gnutella TCP protocol on the localhost socket, if this would seem convenient to the developers.

The Request Processing Block is responsible for the servant reaction to the request – it processes the requests to the servant and sends back the results (if any). The API between the Connection 0 and the Request
Processing Block of the servant obeys the same rules as the API between Connection 0 and the servant’s User Interface Block – it is up to the servant developers to agree on its precise specifications.

The simplest example of the request is the Gnutella file search request – then the Request Processing block performs the search of the local file system or database and returns back the matching filenames (if found) as the search result. But of course, this is not an only imaginable example of the request – it is easy to extend the Gnutella protocol (or to create another one) to deliver the ‘general requests’, which might be used for many purposes other than the file searching.

The User Interface and the Request Processing Blocks together with their APIs (or even the Connection 0 block) can be absent if the Gnutella router (referred to as “GRouter” for convenience in the specification from now on) works without the User Interface or the Request Processing Blocks. That might be the case, for example, when the servant just routes the Gnutella messages, but is not supposed to initiate the searches and display the search results, or is not supposed to perform the local file system or database searches.

The word ‘local’ here does not necessarily mean that the file system or the database being searched is physically located on the same computer that runs the GRouter. It just means that as far as the other servants are concerned, the GRouter provides an access point to perform searches on that file system or database – the actual physical location of the storage is irrelevant. The algorithms presented here were specifically designed in such a way that regardless of the API implementation and its throughput the GRouter might disregard these technical details and act as if the local interface was just another connection, treating it in a uniform fashion. This might be especially important when the local search API is implemented as a network API and its throughput cannot be considered infinite when compared to the TCP connections’ throughput. Thus such a case is just mentioned here and won’t be presented separately – it is enough to remember that the Connection 0 can provide some way to access the ‘local’ file system or database.

In fact, one of the ways to implement the GRouter is to make it a ‘pure router’ – an application that has no user interface or request-processing capabilities of its own. Then it could use the regular Gnutella client running on the same machine (with a single connection to the GRouter) as an interface to the user or to the local file system. Other configurations are also possible – the goal here was to present the widest possible array of implementation choices to the developer.

However, it might be the case that the Connection 0 would be present in the GRouter even if it does not perform any searches and has no User Interface. For example, it might be necessary to use the Connection 0 as an interface to the special requests’ handler. That is, there might be some special requests, which are supposed to be answered by the GRouter itself and would be used by the GNet itself for its own infrastructure-related purposes. One example of such a request is the Gnutella network PING, used (together with its other functions) internally by the network to allow the servants to find the new hosts to connect to. Even if all the GRouter connections are to the remote servants, it might be useful for it to answer the PING requests arriving from the GNet. In such a case the Connection 0 would handle the PING requests and send back the corresponding responses – the PONGs, thus advertising the GRouter as being available for connection.
Still, in order to preserve the generality of the algorithms' description in this specification we assume that all the blocks shown in the diagram are present. This, however, is not a requirement of the invention itself.

Finally, the word 'TCP' in the text and the diagram above does not necessarily mean a regular Gnutella TCP connection, or a TCP connection at all, though this is certainly the case when the presented algorithms are used in the Gnutella network context. However, it is possible to use the same algorithms in the context of other similar 'broadcast-route' distributed networks, which might use different transport protocols – HTTP, UDP, radio broadcasts – whatever the transport layers of the corresponding network would happen to use.

Having said that, we'll continue to use the words 'TCP', 'GNet', 'Gnutella', etc throughout this document to avoid the naming confusions – it is easy to apply the approaches presented here to other similar networks or to other networks that would support operation according to the procedures described.

Now let's go one level deeper and present the internal structure of the Connection blocks shown in Fig. 1.

4. Connection block diagram.

The Connection block diagram is shown in Fig. 2:
The messages arriving from the network are split into three streams:

- The requests go through the Duplicate GUID rejection block first; after that the requests with the ‘new’ GUIDs (not seen on any connection before) are processed by the Q-algorithm block as described in [1]. This block tries to determine whether the responses to these requests are likely to overflow the outgoing TCP connection bandwidth, and if this is the case, limits the number of requests to be broadcast, dropping the high-hop requests. Then the requests, which have passed through it go to the Request broadcaster, which creates N copies of each request, where N is the number of the GRouter TCP connections to its peers (N-1 for other TCP connections and one for the Connection 0). These copies are transferred to the corresponding connections’ hop-layered request buffers and placed there – low-hop requests first. Thus if the total request volume will exceed the connection sending capacity, the low-hop requests will be sent out and the high-hop requests dropped from these buffers.
- The responses go to the GUID router, which determines the connection on which this response should be sent on. Then the response is transferred to this connection's Response prioritization block. The responses with the unknown GUIDs (misrouted or arriving after the routing table timeout) are just dropped.

- The messages used by the Outgoing Flow Control block [1] (OFC block) internally, are transferred directly to the OFC block. These are the 'OFC messages' in Fig 2. This includes both the flow-control 0-hop, 1-TTL PONGs, which are the signal that all the data preceding the corresponding PINGs has already been received by the peer and possibly the 0-hop, 1-TTL PINGs. The former are used by the OFC block for the TCP latency minimization [1]. The latter can appear in the incoming TCP stream if the other side of the connection uses the similar Outgoing Flow Control block algorithm. However, the GRouter peer can insert these messages into its outgoing TCP stream for the reasons of its own, which might have nothing to do with the flow control.

The messages to be sent to the network arrive through several streams:

- The requests from other connections. These are the outputs of the corresponding connections' Q-algorithms.

- The responses from other connections. These are the outputs of the other connections' GUID routers. These messages arrive through the Response prioritization block, which keeps track of the cumulative total volume of data for every GUID, and buffers the arriving messages according to that volume, placing the responses for the GUIDs with low data volume first. So the responses to the requests with an unusually high volume of responses are sent only after the responses to 'normal', average requests. The response storage buffer has a timeout -- after a certain time in buffer the responses are dropped. This is because even though the Q-algorithm does its best to make sure that all the responses can fit into the outgoing bandwidth, it is important to remember that the response traffic has the fractal character [1]. So it is a virtual certainty that from time to time the response rate will exceed the connection sending capacity and bring the response storage delay to an unacceptable value. The 'unacceptable value' can be defined as the delay which either makes the large-volume responses (the ones near the buffer end) unroutable by the peer (the routing tables are likely to time out), or just too large from the user viewpoint. These considerations determine the choice of the timeout value -- it might be chosen close to the routing tables overflow time or close to the maximum acceptable search time (100 seconds or so for the Gnutella file-searching application; this time might be different if the network is used for other purposes).

- The OFC messages are the messages used internally by the Outgoing Flow Control block. These messages can either control the output packet sending (in case of the 0-hop, 1-TTL PONGs -- see [1]) or just have to cause an immediate sending of the PONG in response (in case of the 0-hop, 1-
TTL PINGs). When the algorithm described here is implemented in the context of the Gnutella network, it is useful to remember that the PONG message carries the IP and file statistics information. So since the GRouter’s peer might include the 0-hop, 1-TTL PINGs into its outgoing streams for the reasons of its own – which might be not flow-control-related - it is recommended to include this information into the OFC PONG too. Of course, this recommendation can be followed only if such information is available and relevant (the GRouter does have the local file storage accessible through some API).

All these messages are processed by the ‘RR-algorithm & OFC block’ [1], which decides when and which messages to send; it is this block which implements the Outgoing Flow Control and Fair Bandwidth Sharing functionality described in [1]. It decides how much data can be sent over the outgoing TCP connection, and how the resulting outgoing bandwidth should be shared between the logical streams of requests and responses and between the requests from different connections. In the meantime the messages are stored in the hop-layered request buffers in case of the requests and in the response buffer with timeout in case of the responses.

The OFC messages are never stored – the PONGs are just used to control the sending operations, and the PINGs should cause the immediate PONG-sending. Since it has been recommended in [1] to switch off the TCP Nagle algorithm, this PONG-sending operation should result in an immediate TCP packet sending, thus minimizing the OFC PONG latency for the OFC algorithm on the peer servant. Note that if the peer servant does not implement the similar flow control algorithm, we cannot count on it doing the same - it is likely to delay the OFC PONG for up to 200 ms because of its TCP Nagle algorithm actions. This might result in a lower effective outgoing bandwidth of the GRouter connection to such a host; however, if the 512-byte packets are used, the resulting connection bandwidth can be as high as 25-50 kbits/sec. Still, it is expected that the connection management algorithms would try to connect to the hosts that use the similar flow control algorithms on the best-effort basis.

It should be noted that this approach to OFC PING handling effectively excludes the OFC PONGs from the Outgoing Flow Control algorithm. Since these PONGs are sent at once and thus have the highest priority in the outgoing stream, a DoS attack is possible when the attacker floods its peers with 0-hop, 1-TTL PINGs and causes them to send only PONGs on the connections to the attacker. This can be especially easy to achieve when the attacked hosts have an asymmetric (ADSL or similar) connection.

However, this attack is likely to cause the extremely high latency and/or TCP buffer overflow on the attacked host’s connection to the attacker and result in the connection being closed, which would terminate the attack, as far as the attacked host is concerned. Furthermore, this attack would not propagate over the GNet since by definition it can be performed only with 1-TTL PINGs, which can travel only over 1-hop distance.
5. Blocks affected by the finite message size.

The diagrams presented in the previous sections show the GRouter and the flow control algorithm building blocks and the interaction between them. These diagrams essentially illustrate the flow control algorithm as presented in [1] – no assumptions were made so far about the algorithm changes necessary to allow for the atomic messages of the finite size.

However, Fig. 2 makes it easy to see what parts of the GRouter are affected by the fact that the data flow cannot be treated as a sequence of the arbitrarily small pieces. The affected blocks are the ones that make the decisions concerning the individual messages – requests and responses. Whenever the decision is made to send or not to send a message, to transfer it further along the data stream or to drop – this decision necessarily represents a discrete ‘step’ in the data flow, introducing some error into the continuous-space data flow equations described in [1]. The size of the message can be quite large (at least on the same order of magnitude as the TCP packet size of 512 bytes suggested in [1]). So the blocks that make such decisions implement the special algorithms which would bring the data flow averages to the levels required by the continuous flow control equations.

The blocks that have to make the decisions of that nature and which are affected by the finite message size are shown as circles in Fig. 2. These are the ‘Q-algorithm’ block and ‘RR-algorithm & OFC block’.

The ‘Q-algorithm’ block tries to determine whether the responses to the requests coming to it are likely to overflow the outgoing TCP connection bandwidth, and if this is the case, limits the number of requests to be broadcast, dropping the high-hop requests. The output of the Q-algorithm is defined by the Eq. 13 in [1] and is essentially a percentage of the incoming requests’ data that the Q-algorithm allows to pass through and to be broadcast on other connections. This percentage is a floating-point number, so it is difficult to broadcast an exact percentage of the incoming request data within a finite time interval – there’s always going to be an error proportional to the average request size. However, it is possible to approximate the precise percentage value by averaging the finite data size values over a sufficiently large amount of data. The description of such an averaging algorithm will be presented further in this document.

The ‘RR-algorithm & OFC block’ has to assemble the outgoing packets from the messages in the hop-layered request buffers and in the response buffer. Since these messages have finite size, typically it is impossible (and not really necessary) to assemble the exactly 512-byte packet or to achieve the precise fair bandwidth sharing between the logical streams coming from different buffers as defined in [1] within a single TCP packet. Thus it is necessary to introduce the algorithms that would define the packet-filling and packet-sending procedures in case of the finite message size. These algorithms should desirably follow the general guidelines described in [1], but at the same time they should desirably be able to work with the (possibly quite large) finite-size messages. That means that these algorithms should desirably achieve the general flow control and the bandwidth sharing goals and at the same time should not introduce the major problems themselves. For example, the algorithms should not make the connection latency much higher than the latency that is inevitably introduced by the presence of the large ‘atomic’ messages.
To summarize, the algorithms required in the finite-size message case can be roughly divided into three groups:

- The algorithms which determine when to send the packet and how big that packet should be.

- The algorithms which decide what messages should be placed in the packet in order to achieve the ‘fair’ outgoing bandwidth sharing between the different logical sub-streams.

- The algorithms which define how the requests should be dropped if the total broadcast of all requests is likely to overload the connection with responses.

These algorithm groups are described below:

6. Packet size and sending time.

The Outgoing Flow Control block algorithm [1] suggests that the packet with messages should have the size of 512 bytes and that it should be sent at once after the OFC PONG is received, which confirms that all the previous packet data has been received by the peer. In order to minimize the transport layer header overhead, the G-Nagle algorithm has been introduced. This algorithm prevents the partially filled packets’ sending if the OFC PONG has been already received, but the G-Nagle timeout time $T_N$ (~200 ms) has not passed yet since the last packet sending operation. This is done to prevent the large number of very small packets being sent over the low-latency (<200 ms roundtrip time) links.

This short description of the Outgoing Flow Control block operation leaves out some issues related to the packet size and to the time when it should be sent. The rest of this section explains these issues in detail.

6.1. Packet size.

The packet size (512 bytes) has been chosen as a compromise between two contradictory requirements. First, it should be able to provide a reasonably high connection bandwidth for the typical Internet roundtrip time (~30-35 kbits/sec @ 150 ms), and second, to limit the connection latency even on the low-bandwidth physical links (~900 ms for the 33 kbits/sec modem link shared between 5 connections).

So this packet size value requirement does not have to be adhered to precisely. In fact, different applications may choose a different packet size value or even make the packet size dynamic, determining it in run-time from the channel data transfer statistics and other considerations. What is important is to remember that the packet size growth can increase the connection latency – for example, the modem link mentioned above can have the latency as high as 1,800 ms if the packet size is 1KByte.
Which brings an interesting dilemma: what if the message size is higher than 512 bytes? Even if nothing else is transmitted in the same packet, placing just this one message into the packet can lead to the noticeable latency increase. The Gnutella v.0.4 protocol, for example, limits the message size with at least 64 KBytes (actually the message field size is 4 bytes, so formally the messages can be even bigger). Should the OFC block transmit such a message as a single packet, break it down into multiple packets or just drop it altogether, possibly closing the connection?

In practice the Gnutella servants often choose the third path for the practical reasons, limiting the message size with various numbers (3 KBytes recommended in [1], 256-byte limit for requests used by some other implementations, etc). But here we will consider the most general situation when the maximum message size can be several times higher than the recommended packet size, assuming that the large messages are necessary for the application under the consideration. It is easier to drop the large packets if the GNet application does not require those than to reinvent the algorithms intended for the large messages if it does.

So the first choice to be made is to whether to send a large message in one packet or to split it between the several packets? Note that these ‘packets’ we are discussing here are the packets in terms of TCP/IP, not in terms of the OFC block, which tries to place the OFC PING as a last message in every packet it sends. Since TCP is a stream-oriented protocol that tries to hide its internal mechanisms from the application-level observer, as far as the application code is concerned, this OFC PING is an only semi-reliable sign of the end of the sent data block. (In fact, it is possible that the peer might lose it and the PING retransmission might be required.) For this reason throughout this document the sequence of data bytes between two OFC PINGs, including the second one of them, is referred to as a ‘packet’ – formally speaking, the application-level code cannot necessarily be sure about the real TCP/IP packets used to transmit that data. The packets in terms of TCP/IP protocol are referred to as ‘TCP/IP packets’

When the TCP Nagle algorithm is switched off (as recommended in [1]), typically the send() operation performed by the OFC block really does result in a TCP/IP packet being immediately sent on the wire. However, this is not always the case. It might so happen that for the reasons of its own (the absence of ACK for the previously sent data, the IP packet loss, small data window, or the like) the TCP layer will accept the buffer from the send() command, but won’t actually send it at once. When this buffer will be really sent it might be sent in the same TCP packet with a previous or a subsequent buffer. If the OFC block does not break messages into smaller pieces, this is impossible, since the OFC block would perform no sending operation until the previous one would be confirmed by the PONG from the peer. But if the large message is sent in several 512-byte chunks, it can be the case – several of these chunks can be ‘glued together’ by the TCP layer into a single TCP packet.

On the other hand, when a very large (several kilobytes) message is sent in a single send() operation, the TCP layer can split it into several actual TCP/IP packets, if the message is too big to be sent as a single TCP/IP packet.
So the decision we are looking for here is not final anyway – the TCP layer can change the TCP/IP packets' layout, and the issue here is what would be the best way to do the send() operations, assuming that typically the TCP layer would not change the decisions we wish to make if the Nagle algorithm is switched off.

Assuming for purpose of the next question that the actual TCP/IP packet layout corresponds precisely to the send() calls we make in the GRouter, let's ask ourselves a question: what are the advantages and disadvantages of both approaches?

On one hand, sending a big message in a single packet would undoubtedly result in higher connection bandwidth utilization when the OFC algorithm is used. However, this might cause the connection latency to increase and open the way for the big-packet DoS attack. Besides, if the higher connection bandwidth utilization is desirable, it is better to do it in a controlled way – by increasing the packet size from 512 bytes to a higher value instead of relying on the randomly arriving big messages to achieve the same effect. It is also important to remember that in many cases the higher bandwidth utilization can have a detrimental effect on the concurrent TCP streams (HTTP up/downloads, etc) on the same link, so it might be undesirable in the first place.

So the recommended way is to split the big message into several packets. But this might have some negative consequences in the context of the existing network, too – for example, some old Gnutella clients seemed to expect the message to arrive in the single packet and the message that has been split into several packets might cause them to treat it incorrectly. Even though these clients are obviously wrong, if there are enough of these in the network, it might be a cause for concern. Fortunately this is just a backward compatibility problem in the existing Gnutella network, and in this case there is another way to deal with such a problem. Since the Gnutella network message format is clearly documented, it might be a good idea to split the big incoming message into several smaller messages of \( \leq 512 \) bytes each.

In fact, such a solution (when it is possible) is an ideal variant of dealing with big messages. When the big message is split into several messages, it makes it possible to send other messages between these on the same TCP connection – not just on the same physical link, as it is the case when the big message is just split into several TCP packets. This would minimize the latency not only for the different connections on the same physical link, but also for the connection used to transmit such a message. For example, the requests being sent on the same connection would not have to wait until the end of the big message transfer, but could be sent 'in the middle' of such a message. As a side benefit, the attempt to perform the 'big message' DoS attack would be thwarted by the Response prioritization block in Fig. 2. The resulting sub-messages with a high response volume would be shifted to the response buffer tail, where they might be even purged by the buffer timeout procedure if the bandwidth would not be enough to send those.

To summarize, the GRouter should try to break all the messages into small \((\leq 512 \text{ byte})\) messages. If this is not possible, it should send the big unbreakable messages in the \((\leq 512\text{-byte sending operations (TCP packets)})\), unless it is de facto impossible due to the backward compatibility issues on the network. Since it is impossible to append the OFC PING to such a packet (it would be in the middle of the message), these TCP packets should be sent without waiting for the OFC PONGs, and the OFC PING should be appended to the last
packet in a sequence. The GRouter should desirably never send the messages with a size bigger than some limit (3 Kbytes or so, depending on the GNet application), dropping these messages as soon as they are received.

The related issue is the GRouter behavior towards the messages that cause the packet overflow - when the message to be placed next into the non-empty packet by the RR-algorithm makes the resulting packet bigger than 512 bytes. Several actions are possible:

First, the message sending can be postponed and the packet of less than 512 bytes can be sent.

Second, the message can be placed into the packet anyway, and the packet, which is bigger than 512 bytes can be sent.

And third, n exactly 512-byte packets (where n>=1) can be sent with the last message head and no OFC PINGs; then a packet with the last message tail and OFC PING should immediately follow this packet (or packets).

The general guideline here is that (backward compatibility permitting) the average size of the packets sent as the result should be as close to 512 as possible. If we designate the volume of the packet before the overloading message as V1, the size of this message as V2, and the desired packet size (512 bytes in our case) as V0, we will arrive to the following average packet size values Vavi:

In the first case,

\[ V_{a1} = V1 \]

In the second case,

\[ V_{a2} = V1 + V2 \]

And in the third case,

\[ V_{a3} = (V1 + V2) / (n + 1) \]

So whenever this choice presents itself, all three (or more, if V2 is big enough to justify n>1) Vavi values should be calculated, and the method, which gives us the lowest value of \( \text{abs}(V_{a1} - V0) \) (or some other metrics, if found appropriate) should be used.
6.2. Packet sending time.

It has been already mentioned that the packet (in OFC terms) should desirably not be sent before the OFC PONG for the previous packet ‘tail PING’ arrives. That PONG shows that the previous packet has been fully received by the peer. Furthermore, if the PONG arrives in less than 200 ms after the previous sending operation and there’s not enough buffered data to fill the 512-byte packet, this smaller packet should not be sent before this 200-ms timeout expires (G-Nagle algorithm).

However, these requirements are introduced by the OFC (Outgoing Flow Control) block [1] for the latency minimization purposes and define just the earliest possible sending time. In reality it might be necessary to delay the packet sending even more. The reason for this is that the sent packet size and its PONG echo time are the only criteria that can be used by the upstream algorithm blocks (RR-algorithm and the Q-algorithm) to evaluate the channel bandwidth, which is needed for these blocks to operate. No other data is available for that purpose, and even though it might be possible to gather various channel statistics, such data would be extremely noisy and unreliable. Typically multiple TCP streams share the same connection and it is very difficult to arrive to any meaningful results under such conditions. In fact, in the absence of the bandwidth reservation block (like the one defined by the RSVP protocol) in the TCP layer of the network stack this task seems to be just plain impossible. Any amount of statistics can be made void at any moment by the start of the FTP or HTTP download by some other application not related to the GRouter.

When the packets have the full 512-byte size, it is possible to approximate the bandwidth by the equation:

\[
B = \frac{V_0}{Trtt},
\]

where \( B \) is the bandwidth estimate, \( V_0 \) is the full packet size (512 bytes) and \( Trtt \) is the GNet one-hop roundtrip time, which is the interval between the OFC packet sending time and the OFC PONG (reply to the ‘trailer’ PING of that OFC packet) receiving time.

Even though this bandwidth estimate may not be very accurate under all circumstances and may vary over a wide range in certain circumstances, it is still possible to use it. It can be averaged over the large time intervals (in case of the Q-algorithm) or used indirectly (when the bandwidth sharing is calculated in terms of the parts of packet dedicated to the different logical sub-streams in case of the fair bandwidth-sharing block).

The situation becomes more complicated when there’s not enough data to fill the full 512-byte packet at the moment when this packet can be already sent from the OFC block standpoint. Let us consider the model situation when the total volume of requests passing through the GRouter is negligible (each request causes multiple responses in return). Then the connection bandwidth would be used mostly by the responses, and the Q-algorithm would try to bring the bandwidth used by responses to the \( B/2 \) level, as shown in Fig. 3:
In order to do that, the Q-algorithm is supposed to know the bandwidth \( B \) – otherwise it cannot judge how many requests should it broadcast in order to receive the responses that would fill the \( B/2 \) part of the total bandwidth. Let’s say that somehow this goal has been reached and the data transfer rate on the channel is currently exactly \( B/2 \). Now we want to verify that this is really the case by using the observable traffic flow parameters and maybe make some small adjustments to the request flow if \( B \) is changing over time. Would the number of requests’ data be enough to fill the ‘empty’ part of the bandwidth in Fig. 3, then (5) could be used to estimate the total bandwidth \( B \). Then the packet volume would be more or less equally shared between the requests and responses, and we should try to reach exactly the same amount of request and response data in the packet by varying the request stream. (Not the request stream in this packet, but the one in the opposite direction, which is not shown in Fig. 3.)

But since there are virtually no requests, in the state of equilibrium (constant traffic stream and roundtrip time) we have to estimate the full bandwidth \( B \) using just the size of the packets with back-traffic (response) data \( V \) and the GNet roundtrip time \( T_{rtt} \).

The problem is, it is very difficult to estimate the total bandwidth from that data. If we assume that we are sending packets as soon as the OFC PONG arrives and that the sending rate is \( b \), we arrive to the following relationship between \( V \), \( T_{rtt} \) and \( b \):

\[
(6) \quad V = b \times T_{rtt}
\]

Now, how should we arrive to the conclusion about whether \( b \) is less, more or equal to \( B/2 \) from that information, if we have no idea what is the value of \( B \)? And we need this answer in order to figure out whether to throttle down the broadcast rate, to increase it or to leave it at the same level (Eq. 10 in [1]).

One might expect that if we can effectively change the bandwidth allocation by varying the volume of data in the full (512-byte) packet, we might try to do the same in case of the partially filled packet and estimate the bandwidth \( B \) as \( B_{app} = b \times V_0 / V \). However, such an approach may not always be successful. The reason for this is that in case of the full packet, its expected average roundtrip time \( <T_{rtt}> \) does not change when the
packet internal layout is changed; so the response sending rate $b$ is actually related to the full connection bandwidth $B$ by the equation:

$$
(7) \quad b = B \times \frac{V}{V_0}
$$

This equation can be used only if the packet is full and $V$ is not the packet size, but the size of the response data in this 512-byte packet.

On the contrary, if the packet is just partially filled and $V$ is its total size, its expected roundtrip time $Trtt$ is not constant and might depend on the packet size $V$. For example, if the connection is sufficiently slow, $Trtt$ might be proportional to $V$. Then the value of $B$ estimated from (7) as $b \times V_0/V$ (when $V$ is the total packet size) would give the results that are dramatically different from any reasonably defined total bandwidth $B$ – this estimate would go to infinity as the packet size $V$ goes to zero! In fact, even the state of the equilibrium itself as defined above (constant $V$, $b$ and $Trtt$) would be impossible in this case – if $Trtt=V/B$ and $V=b \times Trtt$, then for a constant-rate response stream $b$

$$
(8) \quad V(t+Trtt) = (b/B) \times V(t),
$$

which means that for every response rate $b$ lower than the actual connection bandwidth $B$, the values of $V$ and $Trtt$ would decline exponentially over time until the G-Nagle timeout or the zero-data roundtrip time is reached. That might result in the very small values of $V$ (packet size) and huge bandwidth estimate values, possibly causing the self-sustained uncontrollable oscillations of the request and response traffic defined by the Q-algorithm.

For these reasons, it is highly desirable to introduce a controlled delay into the packet sending procedure in order to evaluate the target channel bandwidth $B$ when the actual traffic sending rate $b$ is less than $B$. This delay provides an only way to stabilize the packet size $V$ at some reasonable level ($V=V_0$ and $V$ does not go to zero) when the actual traffic rate $b$ is less than $B$ (defined by (5), if it would be possible to send the full 512-byte packets. Actually this 'theoretical' value of $B$ is not directly observable when the total traffic is low and $V<V_0$. The very fact that $B$ is not directly observable under these conditions is what has caused our problems to begin with.)

This delay value (wait time) $Tw$ is defined as the extra time that should pass after the OFC PONG arrival time before the packet should actually be sent and is calculated with the following equations:

$$
(9) \quad Tw = Trtt \times (V_0 - V)/V, \quad \text{if} \quad V_0/2 \leq V \leq V_0
$$

$$
(10) \quad Tw = Trtt, \quad \text{if} \quad V < V_0/2
$$

$$
(11) \quad Tw = 0, \quad \text{if} \quad V > V_0.
$$

The equations (9-11) assume that the G-Nagle algorithm is not used ($Trtt + Tw \geq TN; TN=200$ ms); if this is not the case, the G-Nagle algorithm takes priority.
(12) \[ T_w = T_N - T_{rtt}, \quad \text{if} \quad T_{rtt} + T_w(\text{from 9-11}) < T_N \text{ and } V < V_0 \]

It is easy to see that in case of the full packet (\( V=V_0 \) and \( b=B \)), \( T_w=0 \). The delay is effectively used only when it is necessary to do the bandwidth estimate in case of the low traffic (\( b=B \)). The equation (10) caps the \( T_w \) growth in case of the small packet size.

Then the total theoretical connection bandwidth \( B \) is estimated by its approximate value \( B_{appr} \), which is calculated as:

(13) \[ B_{appr} = \frac{V_0}{T_{rtt}(V)}, \quad \text{if} \quad V \leq V_0 \]
(14) \[ B_{appr} = \frac{V}{T_{rtt}(V)}, \quad \text{if} \quad V > V_0 \]

The full description of reasons that led to the introduction of \( T_w \) and \( B_{appr} \) in the form defined by (9-14) is pretty lengthy and is outside the scope of this document. However, it should be said that unfortunately it does not seem possible to have a precise estimate of \( B \) even when a delay is used. The error of \( B_{appr} \) when compared to \( B \) as defined by (5) depends on many factors. Shortly speaking, different forms of the functional relationship between \( T_{rtt} \) and \( V \) (the form of the \( T_{rtt}(V) \) function) can influence this error significantly. At the same time, it is very difficult to find the actual shape of the \( T_{rtt}(V) \) function with any degree of accuracy under the real network conditions, and this function’s shape can change faster than the statistical methods would find the reasonably precise shape of this function anyway.

So the equations (9-14) represent the result of the attempts to find a bandwidth estimate that would produce a reasonably precise value of \( B_{appr} \) in the wide range of the possible \( T_{rtt}(V) \) function shapes. The analysis of different cases (different \( T_{rtt}(V) \) function shapes, G-Nagle influence, etc) shows that if the Q-algorithm tries to bring the value of \( b \) to the \( \rho \ast B \) level, the worst possible estimate of \( B \) using the equations (9-14) results in a convergence of \( b \) to:

(15) \[ b \to \rho \ast B / \sqrt{\rho}, \]

which for the \( \rho = 0.5 \) suggested in [1] results in \( b \) actually converging to the level \( 0.707 \ast B \) instead of \( 0.5 \ast B \) when the request traffic is nonexistent (as in Fig 3). Naturally, in the real network at least some request traffic would be present, bringing the actual total traffic closer to its theoretical limit \( B \) (as defined in (5)) and making the error even smaller. However, if this 40% increase in the response traffic happens to be a problem under some real network conditions because of the fractal character of the traffic and would cause the frequent response overflows, it is always possible to use smaller values of \( \rho \). For example,

(16) \[ b \to 0.55 \ast B, \quad \text{if} \quad \rho = 0.3 \]

even in the biggest possible error case.
Just to illustrate the equations (9-14) operation, let’s have a look at the same shape of the $\text{Trtt}(V)$ function as the one considered earlier: $\text{Trtt} = V / B$.

Then the equation (13) would give us the following bandwidth approximation:

\[(17) \quad \text{Bappr} = B * V_0 / V,\]

and, the Q-algorithm would bring the response traffic rate to

\[(18) \quad b = 0.5 * \text{Bappr} = 0.5 * B * V_0 / V \quad \text{(if rho = 0.5)}\]

The response stream with this rate would, in turn, result in the packets of size

\[(19) \quad V = b * (\text{Trtt} + T_w) = b * \text{Trtt} * V_0 / V \quad \text{(after we substitute Tw from (9))}\]

Now, since $\text{Trtt} = V / B$, we arrive to

\[(20) \quad V = b * V_0 / B.\]

Combining this with (18), we receive

\[(21) \quad V^2 = b * V_0^2, \quad \text{or} \quad V = V_0 / \text{sqr}(2),\]

and,

\[(22) \quad b = 0.5 * B * \text{sqr}(2) = 0.707 * B\]

First, this result verifies the correctness of substitution of equation (9) for $T_w$ into (19) and the correctness of using the equation (13) as the basis for (17). And second, it shows that in that case the state of the equilibrium (constant $V$, $b$ and $\text{Trtt}$) is achievable for the traffic and the response bandwidth error is exactly the one suggested by the equation (15). (This example uses a pretty ‘bad’ shape of the $\text{Trtt}(V)$ function from the Bappr error standpoint – we could have analyzed many cases with the lower or even nonexistent Bappr error, but it is useful to have a look at the worst case).

Finally it should be noted that the equations (9-14) contain only the packet total size and roundtrip times and say nothing of whether the packet carries the responses, the requests or both. Even though we used the model situation of nonexistent request traffic (Fig. 3) to illustrate the necessity of this approach to the bandwidth estimate, the same equations should also be used in the general case, when the packet carries the traffic of both types. In fact, it can be shown that the error of the Bappr estimate approaches zero regardless of the $\text{Trtt}(V)$ function shape when the total packet size $V$ (responses and requests combined) approaches $V_0$ (512 bytes).
7. Packet layout and bandwidth sharing.

The packet layout and the bandwidth sharing between the sub-streams are defined by the Fairness Block algorithms [1]. The Fairness Block goal is twofold:

- To make sure that the outgoing connection bandwidth available as a result of the outgoing flow control algorithm operation is fairly distributed between the back-traffic (responses) intended for that connection and the forward-traffic (requests) from the other connections (the total output of their Q-algorithms).

- To make sure that the part of the outgoing bandwidth available for the forward-traffic broadcasts from other connections is fairly distributed between these connections.

The first goal is achieved by 'softly reserving' some part of the outgoing connection bandwidth $G_i$ for the back-traffic and the remainder of the bandwidth -- for the forward-traffic. The bandwidth 'softly reserved' for the back-traffic is $B_i$ and the bandwidth 'softly reserved' for the forward-traffic is $F_i$:

![Diagram](image)

Fig. 4. The bandwidth reservation layout.

'Softly reserved' here means, for example, that when, for whatever reason, the corresponding stream does not use its part of the bandwidth, the other stream can use it, if its own sub-band is not enough for it to be fully sent out. But if the sum of the desired back- and forward-streams to be sent out exceeds $G_i$, each stream is guaranteed to receive at least the part of the total outgoing bandwidth $G_i$ which is 'softly reserved' for it ($B_i$ or $F_i$) regardless of the opposing stream bandwidth requirements. For brevity's sake, from now on, we will actually mean 'softly reserved' when we will apply the word 'reserved' to the bandwidth.

In Fig. 4, the current back-traffic $b_i$ is shown to be two times less than $B_i$, since Q-algorithm tries to keep the back-stream at that level; however, it can fluctuate and be much less than $B_i$ if the requests do not generate a lot of back-traffic, or temporarily exceed $B_i$ in case of the back-traffic burst. If $b_i\leq B_i$, the entire bandwidth above $b_i$ is available for the forward-traffic. If the desired back-traffic exceeds $B_i$, the actual back-traffic $b_i$ can be higher than $B_i$ only if the desired forward-traffic from the other connections $y_i$ is less than $F_i$;
otherwise, the back-traffic fully fills the Bi sub-band and the forward-traffic fully fills the Fi. So the actual forward-traffic stream fo is equal to the desired forward-traffic yi only if either yi<Fi, or yi+bi<Gi; otherwise, fo<i yi and some forward-traffic (request) messages have to be dropped.

7.1. Simplified bandwidth layout.

The method calculates the bandwidth reserved for the back-traffic Bi in [1] (Eq. 24-26) essentially tries to achieve the convergence of the back-traffic bandwidth Bi to some optimal value:

\[(23) \quad <Bi> \rightarrow <Gi - 0.5*foi>\]

This optimal value was chosen in such a way that it would protect the forward-traffic (requests from other connections) in case of the back-traffic (response) bursts – the bandwidth reserved for the forward-traffic (Fi=Gi-Bi) should be no less than half of the average forward traffic <foi> on the connection. Thus the back-traffic bursts cannot significantly decrease the bandwidth part used by the forward traffic or completely shut off the forward traffic data flow. Similarly, the back-traffic is protected from the forward-traffic bursts – from the equation (23) it is clear that Bi>=0.5*Gi, so at least half of the connection bandwidth is reserved for the back-traffic in any case.

However, in case of the finite message size, the equation (23) has one problem. Let us consider a ‘GNet leaf’ structure, consisting of a GRouter and a few neighbors, none of which are connected to anything besides the GRouter. Such a configuration is shown in Fig. 5:

![Fig. 5. The ‘GNet leaf’ configuration.](image)

Here ‘Connection i’ connects this ‘leaf’ structure to the rest of the GNet. We will be interested in the traffic passing through this connection from right to left – from the ‘leaf’ to the GNet. The GRouter Fairness Block controls this traffic. Such a configuration is typical for the various ‘GNet reflectors’, which act as an interface to the GNet for several servants, or for the GRouter working in a ‘pure router’ mode. Then the GRouter has no user interface and no search block of its own and just routes the traffic for another servant (or
several servants). Typically that configuration would result in a very low volume of request data passing through this ‘Connection i’ from right to left, since the ‘leaf’ has just a few hosts.

Because of this, the equation (23) in the GRouten block fairness might bring the value of $Bi$ very close to $Gi$ for that connection. To be precise, the stable value of $Fi$ would be:

$$Fi = 0.5 \times \langle foi \rangle,$$

where $\langle foi \rangle$ is a very low average forward-traffic sending rate. In the continuous-traffic model $Fi=const$, since this low sending rate $\langle foi \rangle$ is represented by the fairly constant low-volume data stream. The equation (23) convergence time (defined by the Eq. 15 in [1]) is irrelevant in that case.

The atomic messages (requests) of the finite size change this situation dramatically. Then every request represents a traffic burst of the very high instant magnitude (mathematically, it can be described as the delta-function – the infinite-magnitude burst with the finite integral equal to the request size). The equation (23) will try to average the sending rate, but since it has a finite convergence (averaging) time, in case the average interval between finite-size requests is bigger than the convergence time, the plot of $Fi$ versus time will look like this:

![Diagram showing the finite-size request rate averaging.](image)

Fig. 6. The finite-size request rate averaging.

The plot in Fig 6 makes it clear that if the average interval between requests is bigger than the equation (23) convergence time, the bandwidth $Fi$ reserved for the requests can be arbitrarily small at the moment of the next request arrival. Since the equation (23) convergence time is not related to the request frequency (which might be determined by the users searching for files, for example), the small frequency of the requests leads to the small value of $Fi$ when the request does arrive on the connection to be transmitted.

So when the request arrives, the bandwidth reserved for it might be very close to zero. If the back-traffic from the ‘leaf’ does not have a burst at that moment, it would occupy just about one half of the available bandwidth $Gi$, and the request transmission would not present any problem. But if the back-traffic experiences a burst, the bandwidth available for the request transmission would be just a very small reserved bandwidth $Fi$. 
Thus the time needed to transmit the finite-size request might be very large, even if the request would not be atomic. (In that case the start of the request transmission would gradually lower the Bi and this request transmission would take an amount of time comparable to the convergence time of the equation (23)).

However, since the request is atomic (unbreakable) and cannot be sent in small pieces between the responses on the same connection, the delay might be even bigger. In order to make sure that the sending operation does not exceed the reserved bandwidth, the sending algorithm has to ‘spread’ the request-sending operation over time, so that the resulting average bandwidth would not exceed a reserved value. Since from the sending code standpoint the request is sent instantly in any case, the ‘silence period’ of the $T_s=V_r/F_i$ length would have to be observed after the request-sending operation in order to achieve that goal, where $V_r$ is the request size. This ‘silence period’ can be arbitrarily long, because equation (23) decreases $F_i$ in an exponential fashion as the time since the last request arrival keeps growing. If the next request to be sent arrives during this ‘silence period’ (which is quite likely when $T_s$ grows to infinity), this new request either has to be kept in the fairness block buffers until the back-traffic burst ends, or to be just dropped.

Neither outcome is particularly attractive – on one hand, it is important to send all the requests, since the ‘Connection i’ is the only link between the ‘leaf’ and the rest of the GNet. And on the other hand, it is intuitively clear that the latency increase due to the new request being buffered for the rest of the ‘silence period’ is not necessary. After all, the request traffic from the ‘leaf’ is very low, and it would seem that sending all the requests without delays should not present any problem.

So the fairness block behavior seems be counterintuitive: if it is intuitively clear that the requests can be sent at once, why the equation (23) does not allow us to do that? To explain that, it should be remembered that the exponential averaging performed by the differential equation (23) (equation (26) in [1]) was designed to handle the continuous-traffic case. This averaging method assumes that the traffic being averaged consists of a very large number of very small and very frequent data chunks, which is clearly not the case in the example above. When the time interval between the requests exceeds the averaging (equation (23) convergence) time, these equations cease to perform the averaging function, which results in the negative effects that we could observe here.

Besides, the Fairness Block equations were designed to protect the average forward-traffic from the back-traffic bursts and other way around. These equations do nothing to protect the forward-traffic bursts, since it was assumed that it is enough to reserve the forward-traffic bandwidth that would be close to the average forward-traffic-sending rate. This approach really works when the forward-traffic messages (requests) are infinitely small. However, as the averaging functionality breaks down with the growth of the interval between requests, and each request is a traffic burst, nothing protects this request from the simultaneous burst in the back-traffic stream, resulting in the latency increase and possibly in the request loss.

Thus it is clear that the finite-message case presents a very serious problem for the Fairness Block, and something should be done to deal with the situations like the one presented above. In principle, it might be
possible to extend the Fairness Block equations to handle the case of the ‘delta-function-type’ (non-continuous) traffic. However, such an approach is likely to be complicated, so here we suggest a radically different solution.

Let us make both reserved sub-bands \((B_i\) and \(F_i\) fixed:

\[
\begin{align*}
(25) & \quad F_i = G_i / 3, \\
(26) & \quad B_i = 2G_i / 3
\end{align*}
\]

and compare the resulting bandwidth layout with the ‘ideal’ layout in an assumption that such a layout really does exist and can be found.

The solution presented in (25,26) is not an ideal one – it does not take into consideration the different network situations, different relationships between the forward- and backward-traffic rates and so on. Thus it is expected that in some cases such a bandwidth layout would result in a smaller connection traffic than the ‘ideal’ layout, effectively limiting the ‘request reach’: the servants would be able to reach fewer other servants with their requests and would receive less responses in return.

Let’s check the maximal theoretical throughput loss for the back- and forward-traffic streams in case of the fixed bandwidth layout (25,26).

The biggest possible average back-traffic is

\[
(27) \quad <b_{\text{imax}} > = 0.5 * <G_i >,
\]

and the average fixed-bandwidth traffic is

\[
(28) \quad <b_i > = 0.5 * B_i = G_i/3.
\]

Thus the worst theoretical response throughput loss is about 33 \%. However, the fixed bandwidth layout is going to be used together with the bandwidth estimate algorithm described in section 6.2 of this document. That algorithm is capable of increasing the back-traffic by a factor of 0.707 (Eq. (15) with \(\rho=0.5\)) in some cases, so these errors might even cancel each other, possibly resulting in an average back-traffic \(<b_i >-0.47*G_i\), which is pretty close to an ideal value.

The biggest possible average forward-traffic is

\[
(29) \quad <f_{\text{imax}} > = <G_i >.
\]

In case of the fixed bandwidth the average forward traffic is limited by the average back-traffic \(<f_{\text{oi}} >-<G_i-b_i >\). However, since the average back-traffic should not take more than 1/3 of the whole bandwidth (Eq. (28)), then
(30) \[ <\text{foi}> = \frac{2^*<\text{Gi}>}{3}, \]

which represents a 33% theoretical request throughput loss.

At the first glance, one might expect that in the very worst case (back-traffic errors cancel and \(<\text{bi}>=0.47^*<\text{Gi}>\)), the average forward-traffic would be limited by the expression \(<\text{foi}> = 0.53^*<\text{Gi}>\), meaning that a 47% request throughput loss is possible. However, for the equation (15) to be applicable, the total traffic \(\text{bi}+<\text{foi}>\) has to be less than \(\text{Gi}\). But if this is the case, there are not enough requests to fill the full available bandwidth \((\text{Gi}-\text{bi})\) anyway. So then the fixed bandwidth layout approach does not limit the request stream-sending rate and as far as the forward stream is concerned, there are no disadvantages introduced by the fixed bandwidth layout at all.

Thus the worst possible throughput loss for both back- and forward-traffic is about 33% versus the 'ideal' bandwidth-sharing algorithm, assuming that such an algorithm exists and can be implemented. This throughput loss is not very big and is fully justified by the simplicity of the fixed bandwidth sharing. It is also important to remember that this number represents the worst throughput loss — in real life the forward-traffic throughput loss might be less if the response volume is low. Then \(\text{bi} < \frac{\text{Bi}}{2}\) and the bandwidth available to the forward-traffic is going to be bigger. All these considerations make the fixed bandwidth sharing as defined by (25,26) the recommended method of bandwidth sharing between the request and response sub-streams.

7.2. Packet layout.

In practice the value of \(\text{Gi}\) can fluctuate with each packet and is not known before the packet is actually sent, making the values of \(\text{Bi}\) and \(\text{Fi}\) also hard to predict. This makes it very difficult to fulfill the bandwidth reservation requirements (25,26) directly, in terms of the data-sending rate. The relationship between the bandwidths of the forward- and back-streams has to be maintained indirectly, by varying the amount of the corresponding sub-stream data placed into the packet to be sent. Naturally, the presence of the finite-size atomic messages complicates this process further, making the precise back- and forward-data ratio in the packet hard to achieve.

Let us start with a simpler task and imagine that the traffic can be treated as a sequence of the arbitrarily small pieces of data and see how the bandwidth sharing requirements (25,26) would look in terms of the packet layout.

The packet to send is assembled from the continuous-space data buffers (Hop-layered request buffers and a Response buffer in Fig. 2) when the packet-sending requirements established in section 6.2 have been fulfilled. To simplify the task even more, let’s imagine that we have a single request buffer, so the packet is filled by the data from just two buffers — the request and the response one.
If the summary amount of data in both buffers does not exceed the full packet size \( V_0 \) (512 bytes), the packet-filling procedure is trivial – both buffers’ contents are fully transferred into the packet, and the resulting packet is sent, leaving us with empty request and response buffers. In terms of the bandwidth usage, it corresponds to the case of the bandwidth non-overflow, and in case the total amount of data sent is even less than 512 bytes, the equations (9-11) show that an additional wait time is required before sending such a packet. Which means that the bandwidth is not fully utilized - we could increase the sending rate by bringing the waiting time \( T_w \) to zero and filling the packet to its capacity, if we’d have more data in request and response buffers.

Looking at the bandwidth reservation diagram in Fig. 4, we see that in such a case (\( b_i + f_i = < G_i \)) the bandwidth reservation limits \( B_i \) and \( F_i \) are irrelevant. These are the ‘soft’ limits and have to be used only if the sum of the desired back- and forward-traffic sending rates \( b_i \) and \( y_i \) exceeds the full bandwidth \( G_i \).

Of course, even though \( B_i \) is not used to limit the traffic, it still has to be communicated to the Q-algorithm of that connection so that it could control the amount of request data it passes further to be broadcast. In order to find the \( B_i \), the total channel bandwidth \( G_i \) has to be approximated by the \textit{Bappr} found from (13). Then the \( B_i \) estimate is found from (26) as

\[
(31) \quad B_i = 2 \times \text{Bappr} / 3 = 2/3 \times V_0 / \text{Trtt}.
\]

Naturally, this can be done only postfactum, after the packet is sent and its PONG echo is received from the peer, but that does not matter – the Q-algorithm equations [1] are specifically designed to be tolerant to the delayed and/or noisy input.

Now let’s consider the case when the summary amount of data in the request and the response buffers exceeds the desired packet size \( V_0 \) (512 bytes). Since we are still working in the continuous-traffic model, it is clear that the packet size should be exactly \( V_0 \) and the wait time \( T_w \) should be zeroed. And now we face a question – how much data from each buffer should be placed into the packet in order to make the packet of exactly \( V_0 \) size and satisfy the bandwidth reservation requirement (25,26)?

Let us designate the amount of forward (request) data in the packet as \( V_f \) and the amount of back-data (responses) as \( V_b \). Obviously,

\[
(32) \quad V_f + V_b = V_0.
\]

After the packet PONG echo returns and the total bandwidth \( G_i \) estimate \textit{Bappr} is calculated from (14), it will be possible to find the value of \( B_i \) from (31) as

\[
(33) \quad B_i = 2/3 \times V_0 / \text{Trtt},
\]

and the value of \( F_i \) as
(34) \[ F_i = B_{appr} / 3 = 1/3 \cdot V_0 / \text{Trt}. \]

At the same time (after the PONG echo is received) it will be possible to find the sending rates of the forward- and back-traffic as

\begin{align*}
(35) & \quad f_{oi} = V_f / \text{Trt} \quad \text{and} \\
(36) & \quad b_i = V_b / \text{Trt},
\end{align*}

after which we would be able to see whether the values of \( f_{oi} \) and \( b_i \) exceed the reserved bandwidth values \( F_i \) and \( B_i \) or not. However, that would be too late – we need this answer before we send the packet in order to determine the desired values of \( V_f \) and \( V_b \) for it. Fortunately, even before we send the packet, from (34) and (35) it is clear that

\begin{align*}
(37) & \quad f_{oi} / F_i = 3 \cdot V_f / V_0, \\
\text{and from (33) and (36)} & \\
(38) & \quad b_i / B_i = 3/2 \cdot V_b / V_0,
\end{align*}

which means that if \( b_i = B_i \) and \( f_{oi} = F_i \), then

\begin{align*}
(39) & \quad V_f = V_0 / 3 \\
(40) & \quad V_b = 2 \cdot V_0 / 3
\end{align*}

So using (39,40) we can determine whether the bandwidth reservation requirements (25,26) will be satisfied even before we send the packet. It should be remembered, though, that the bandwidth reservation requirements (25,26) are ‘soft’. That is, we can have \( V_f \) or \( V_b \) exceeding the value defined by (39) or (40), provided that the opposite stream can be fully sent (the amount of data in its Fig. 2 buffer is less than the value defined by the equation (40) or (39), correspondingly). First, we try to put \( V_f \) and \( V_b \) bytes of requests and responses into the packet. If some buffer does not have enough data to fully fill its \( V_x \) packet part, then the data from the opposite buffer can be used to pad the packet to \( V_0 \) size, provided that there’s enough data available in this opposite buffer.

Then, after the packet is sent and its PONG OFC echo returns, we should calculate the actual value of \( B_i \) for the Q-algorithm, using the same equation (31) that we use for the packet with size \( V < V_0 \).

Now that we have the bandwidth reservation requirements (25,26) translated into the packet volume terms (39,40), we can abandon the continuous-traffic assumption and consider the case of the finite-size atomic messages.
In this case the request and the response buffers contain the finite-size messages, which can be either fully placed into the packet, or left in the buffer (for now, we’ll continue assuming that there’s just one request buffer – the multiple-buffer case will be considered later). The buffers are already prioritized according to the request hop (in case of the hop-layered request buffer) or according to the summary response volume (in case of the response buffer). Thus the packet to be sent might contain several messages from the request buffer head and several messages from the response buffer head (either number can be zero).

Here the ‘packet’ means a sequence of bytes between two OFC PINGs – the actual TCP/IP packet size might be different if the algorithm presented in section 6.1 (equations (2-4)) splits a single OFC packet into several TCP/IP ones. Again, we can have two situations – when the summary amount of data in both buffers does not exceed the packet size V0 (512 bytes) and when it does.

If both buffers can be fully placed into the packet, there are no differences between this situation and the continuous-traffic space case at all. Since we are fully sending all the available data in one packet, it does not matter whether it is a set of finite-size messages or a continuous-space volume of data – we are not breaking the data into any pieces anyway. So we can just apply the continuous-traffic case reasoning and, as a final step, calculate the Bi for the Q-algorithm using (31).

If, however, the summary amount of data in request and response buffers exceeds V0 and the messages are atomic and have the finite size, typically it would be impossible to achieve the precise forward- and backward-data size values in the packet as defined by (39,40). Thus we have to use the approximate values for the Vf and Vb, so that in the long run (when many packets are sent) the resulting data volume would converge to the desired request/response ratio:

\[
\text{Sum}(Vb) / \text{Sum}(Vf) \rightarrow 2, \quad \text{as} \quad \text{Sum}(Vb), \text{Sum}(Vf) \rightarrow \text{infinity}.
\] (41)

In order to achieve that goal, the ‘herringbone stair’ algorithm is introduced:

7.3. ‘Herringbone stair’ algorithm.

This algorithm defines a way to assemble the sequence of packets from the atomic finite-size messages so that in the long run the volume ratio of request and response data sent on the connection would converge to the ratio defined by (41). Naturally, the algorithm is designed to deal with the situation when the sum of the desired request and response sub-streams exceeds the connection outgoing bandwidth Gi, but it should provide a mechanism to fill the packet even when this is not the case.

In order to do that, an accumulator variable acc with an initial value of zero is associated with a connection. At any moment when we need to place another message into the packet, we choose between two candidates (the first messages in the request and response buffers) in a following way:
For both messages the ‘probe’ accumulator values (accF for forward-traffic and accB for back-traffic) are calculated:

\[
\begin{align*}
(42) \quad accB &= acc - Sb, \\
(43) \quad accF &= acc + 2*sf,
\end{align*}
\]

where \( Sb \) and \( Sf \) are the sizes of the first messages in the corresponding (response and request) buffers. Then the values of \( \text{abs}(accB) \) and \( \text{abs}(accF) \) are compared, and the accumulator with the smaller absolute value wins, replaces the old \( acc \) value with its \( accX \) value, and puts the message of type ‘\( X \)’ into the packet. This process is repeated until the packet is filled. If at any moment when the choice has to be made, at least one of the buffers is empty and the \( accB \) or \( accF \) value cannot be calculated, the message from the buffer, which still has the data (if any), is placed into the packet. At the same time the \( acc \) variable is set to zero, effectively ‘erasing’ the previous accumulated data misbalance.

The packet is considered ready to be sent according to the algorithm presented in section 6.1 (equations (2-4)). At that point we exit the packet-filling loop but remember the latest accumulator value \( acc \) – we’ll start to fill the next packet from this accumulator value, thus achieving the convergence requirement (41).

Graphically this process can be represented by the picture, which looks like this:

![Graphical representation of the 'herringbone stair' algorithm.](image)

Fig. 7. Graphical representation of the ‘herringbone stair’ algorithm.

The chart in Fig. 7 illustrates the case when both the request and the response buffers have enough messages, so the accumulator does not have to be zeroed, ‘dropping’ the plot onto the ‘ideal’, \( \frac{1}{2} \)-tangent line. (This dashed line represents the packet-filling procedure in case of the continuous-space data, when the traffic can be treated as a sequence of the infinitely small chunks). The horizontal thick lines represent the responses, and the line length between markers is proportional to the response message size. Similarly the vertical thick lines represent the requests. The thin lines leading nowhere correspond to the hypothetical, ‘probe’ \( accX \) values,
which have lost against the opposite-direction step, since the opposite-direction accumulator absolute value happened to be smaller. Thus every step along the chart in Fig. 7 (moving in the upper right direction) represents the step that was closest to an 'ideal' line with a tangent value of 1/2.

This algorithm has been called the 'herringbone stair algorithm' for an obvious reason – the bigger (losing) $\text{accX}$ value probes (thin lines leading nowhere) resemble the pattern left on the snow when one climbs the hill during the cross-country skiing.

So the basic algorithm operation is quite simple. One fine point, which has not been discussed so far, is the fate of the rest of the data in the request or the response buffer after the packet is sent and it could not accept all the data from the corresponding buffer.

In case of the response buffer the situation is clear: the flow control algorithms try not to drop any responses unless absolutely necessary. That is, unless the response storage delay reaches an unacceptable value (see section 4 for the more detailed explanation of what the 'unacceptable delay value' is). If the time spent by the response in buffer does reach an unacceptable timeout limit, the response buffer timeout handler drops such a response, but this is done in a fashion transparent to the packet-filling algorithms described here. No other special actions are required.

The situation with the request buffer is a bit different. This hop-layered buffer was specifically designed to handle a situation when just a small percentage of the requests in this buffer can be sent on the outgoing connection. The idea was that when the GNet has relatively low response traffic and the Q-algorithm passes all the incoming requests to the hop-layered request buffer, since there's no danger of the response overflow, then the GNet scalability is achieved by the RR-algorithm and an OFC block. This block sends only the low-hop requests out, dropping all the rest and effectively limiting the 'request reach' radius regardless of its TTL value and minimizing the connection latency when the GNet is overloaded.

Since on the average, all incoming and outgoing connections carry the same volume of the request traffic, in this situation (when the RR-algorithm and OFC block take care of the GNet scalability issues) the average percentage of the dropped requests (taken over the whole GNet) is about

\[
\text{(44)} \quad P_{\text{drop}} = \frac{(N-1)}{N},
\]

where $N$ is the average number of the GRouter connections. So with $N=5$ links, it can be expected that on the average just about 20% of the requests in the hop-layered request buffer would be sent out and 80% would be dropped.

In case of the continuous-space traffic, we can just clear the request buffer immediately after the packet is sent. This would bring the worst-case request delay on the GRouter to its minimal value, equal to the interval between the packet-sending operations. Unfortunately this is not always possible in the finite-size message case.
The reason for this is that when the requests are infinitely small, we can expect the following request buffer layout when we are ready to begin assembling the outgoing packet:

Here the buffer contains a very large number of the very small requests, and statistically the requests with every possible hop value would be present. So every time the packet is sent, it would contain all the data with low hops and would not include the buffer tail - the requests with a biggest hop value would be dropped. What is important here is that from the statistical standpoint, it is a virtual certainty that all the requests with very low hop values (0,1,2,...) are going to be sent.

To appreciate the importance of that fact, let us consider the 'GNet leaf' presented in Fig. 5. The 'leaf' servers A, B, C can reach the GNet only through the GRouter. When these servants' requests traverse the 'Connection i' link, they have a hop value of 1. So if the GRouter has the significant probability of dropping the hop=1 requests, it is likely that these servants might never receive any responses from the GNet just because the requests would never reach the GNet in the first place. By the same token, if the GRouter's peer in the GNet (the host on the other side of the 'Connection i') is likely to drop the hop=2 requests, the total response volume arriving back to A, B, C will be decreased. Even if the hosts A, B, C would have other connections to GNet aside from the one to the GRouter, it would still be important to broadcast their requests on the 'Connection i'. Generally speaking, the less is the request hop value, the more important it is to broadcast such a request.

As we move to the finite message size case, we immediately notice two differences: first, the number (though not the total size) of the requests in the hop-layered buffer decreases and the statistical rules might no longer apply. For example, as we start to fill the packet, we might have no requests with hop 0, one request with hop 1, two requests with hop 4 and one request with hop 7. This fact will be important later on, as we move to the multi-source herringbone stair algorithm with several request buffers.

The second difference, which is more important for us here, is that the OFC algorithm might choose to send the packet containing only the responses. Let's have another look at Fig. 7 and imagine ourselves that all the messages there (the thick lines between the markers) are bigger than V0 (512 bytes). Then every such message will be sent as a single OFC packet (and maybe multiple TCP/IP packets), which would consist of this
big message (request or response) followed by an OFC PING. Essentially, every marker in the Fig. 7 will correspond to the OFC packet sending operation.

Then, if we would clear the request buffer as soon as the response OFC packet is sent, the requests that have arrived since the last packet-sending operation would be dropped and would have precisely zero chance of being sent regardless of their importance in terms of the hop value. In fact, the herringbone stair algorithm can send several ‘response-only’ packets in a row (see the third ‘step’ in Fig 7 – it contains two responses), making it even more probable that the ‘important’ low-hop request would be lost.

This is why it is important to clear the request buffer only after at least a single request is placed into the packet. The graphical illustration of such an approach is presented in Fig. 9:

![Request buffer clearing algorithm.](image)

This is essentially the plot from the Fig. 7, but with ellipses marking the time intervals during which the incoming requests are just added to the request buffer and nothing is removed from it. The chart assumptions are that first, every message is sent in a single OFC packet, and second, that the physical time associated with the plot marker is the moment when the decision is made to include the message, which begins at the marker, into the packet to be sent. That is, the very first marker (at the lower left plot corner) is when the decision is done to send the first message – the request that is plotted as a vertical line on the chart. The small circle surrounding that first marker means that at this point we can clear the request buffer, removing all the other requests from it.

Then we send a response (a horizontal line), but do not clear the request buffer, since we would risk losing the important requests that could arrive there in the meantime. The request buffer is cleared again only after the herringbone stair algorithm decides to send a request and places this request into the packet (the beginning of the second vertical line). Then the request buffer can be reset again, and the ellipse, which covers the whole first ‘step’ of the ‘stair’ in the plot, shows the period during which the incoming requests were being accumulated in the request buffer. At the end of the horizontal line (when the new packet can be sent), all the
requests accumulated during the time covered by the ellipse start competing for the place in the packet, and the process goes on with the request accumulation periods represented by the ellipses on the chart.

Note that the big ellipse that covers the third ‘step’ of the ‘stair’ is essentially a result of the big third request being sent. If the packet roundtrip time is proportional to the packet size, this ellipse might introduce a significant latency into the request-broadcasting process – the next request to be sent might spend a long time in the buffer. Unless the GNet protocol is changed to allow the non-atomic message sending, such situations cannot be fully avoided. On one hand, the third request was obviously important enough to be included into the packet, and on the other hand, the bandwidth reservation requirements do not allow us to decrease the average bandwidth allocated for the responses, and to send the next request sooner. But at least the ‘herringbone stair’ and the request buffer clearing algorithms make sure that the important low-hop requests have the fair high chance to be sent within the latency limits defined by the current bandwidth constraints.

Since the finite-size messages can lead to the OFC packets with size exceeding \( V_0 \) (512 bytes), it might be that we’ll have to use equation (14) instead of (13) to evaluate the bandwidth \( B_i \) if \( V > V_0 \). So instead of equation (31) for \( B_i \) (as it was the case for the continuous-space traffic), the ‘herringbone stair’ algorithm uses the following equations to evaluate the bandwidth \( B_i \) reserved for the back-traffic:

\[
\begin{align*}
B_i &= 2/3 * V_0 / Trt, & \text{if } V \leq V_0, \\
B_i &= 2/3 * V / Trt, & \text{if } V > V_0,
\end{align*}
\]

where \( V \) is the OFC packet size produced by the ‘herringbone stair’ algorithm.

Finally, it should be noted that even when the request buffer clearing algorithm does allow us to remove all the requests from the buffer, this operation should not be performed unless the reset timeout \( Tr \) time (~200 ms) has passed since the last buffer-clearing operation. This timeout is logically similar to the G-Nagle algorithm timeout introduced previously - its goal is to handle the case when the big packets are sent very frequently on the low-roundtrip-time links. Then the fact that the requests are kept in buffer for 200 ms does not noticeably increase the response latency, but might improve the request buffer layout from the statistical standpoint, bringing it closer to the continuous-space layout presented in Fig. 8.

Now that we have fully described the ‘herringbone stair’ algorithm in case of the single request buffer, we can move to the effects introduced by the presence of the multiple GRouter connections and hop-layered request buffers.

7.4. Multi-source ‘herringbone stair’.

When the GRouter connection has multiple request buffers (that is, the GRouter has more than two connections), the basic principles of the packet-filling operations remain the same. The bandwidth still has to be shared between the requests and the responses, the ‘herringbone stair’ algorithm still plots the ‘stair’ pattern if
there's not enough bandwidth to send all the data – the difference is that now the requests have to be taken from several buffers. This is the job of the hop-layered round-robin algorithm introduced in [1] ('RR-algorithm' block in Fig 2.)

The RR-algorithm essentially prioritizes the 'head' (highest priority, low-hop) requests from several buffers, presenting a 'herringbone stair' algorithm with a single 'best' request to be compared against the response. The reasoning behind the round-robin algorithm design was described in [1]; here we just provide a description of its operational principles with an emphasis on the finite request size case.

The hop-layered round-robin algorithm operation is illustrated by Fig. 10:

![Diagram](image)

Fig. 10. Hop-layered round-robin algorithm.

The algorithm queries all the hop-layered connection buffers in a round-robin fashion and passes the requests to the 'herringbone stair' algorithm. Two issues are important:

- No responses with the high hop values are passed until all the requests with the lower hop values are fully transferred from all the request buffers. If some request buffer has just the high-hop requests, it is just skipped by the round-robin algorithm in the meantime.

- Within one hop layer, the RR-algorithm tries to transfer roughly the same amount of data from all buffers that have the requests with the hop value that is being currently processed. In order to achieve that, every buffer has a hop data counter `hopDataCount` associated with it. This counter is equal to the number of bytes in the requests with the current hop value that have been passed to the herringbone stair algorithm from that buffer during the packet-filling operation that is currently underway. Every time the RR-algorithm fully transfers all the current-hop requests from the buffers, all the counters are reset to zero and the process continues from the next buffer (round-robin sequence is not reset).

The current maximal and minimal `hopDataCount` values for all buffers `maxHopDataCount` and `minHopDataCount` are maintained by the RR-algorithm. The request is transferred from the buffer by the RR-algorithm only if this buffer's `hopDataCount` satisfies the following condition:
(47) \( \text{hopDataCount} < \max \text{hopDataCount OR hopDataCount} = \min \text{hopDataCount} \).

If this condition is not fulfilled, the buffer is just skipped and the RR-algorithm moves on to the next buffer. This prevents the buffers with large requests from monopolizing the outgoing request traffic sub-band, which would be possible if the requests would be transferred from buffers in a strictly round-robin fashion.

When the RR-algorithm is used (that is, there is more than one request buffer), the herringbone stair algorithm has to make a choice as to when it should clear all the requests from these several request buffers.

This decision is influenced by pretty much the same considerations as the similar decision in case of the single request buffer (which is described in section 7.3):

- The request buffer should not be cleared before the whole OFC packet is assembled.
- The request buffer should not be cleared more than once per \( T_r \) (~200-ms) time interval.
- The request buffer should not be cleared in such a way that all the requests in it would be dropped before at least one of them is sent – every request must have a chance to compete for the slot in the outgoing packet with the requests from the same buffer.

So the buffer-clearing algorithm presented in section 7.3 is extended for the multiple-buffer situation. The decision to reset the buffers' contents is done for each buffer individually and the buffer can be cleared no sooner than some request from this buffer is included into the outgoing packet by the 'herringbone stair' algorithm.

Of course, this approach might increase the interval between the buffer resets. For example, if some buffer contains a just a single high-hop request, this request can spend a lot of time in the buffer - until some low-hop request arrives there, or until no other buffer would contain the requests with lower hop values. But this is not a big problem – we are mainly concerned with the low-hop requests' latency, since these are the requests, which are typically passed through by the RR- and 'herringbone stair' algorithms. Even if this high-hop request spends a lot of time in its request buffer before being sent, in practice that would most probably mean that multiple other copies of this request would travel along the other GNet routes with little delay. So the delayed responses to that request copy would make just a small percentage of all responses (even if such a request is not dropped), having little effect on the average response latency.

8. Q-algorithm implementation.

The Q-algorithm [1] goal is to make sure that the response flow would not overload the connection outgoing bandwidth, so it limits the request broadcast to achieve this goal, if necessary. Now let us consider the effects that the messages of the finite size are going to have on the Q-algorithm. We are going to have a look at two separate and unrelated issues: Q-algorithm latency and response/request ratio calculations.
8.1. Q-algorithm latency.

The Q-algorithm output is defined by the equation (1) or (52) (Eq. (13) in [1]). This equation essentially defines the percentage of the forward-traffic (requests) to be passed further by the Q-algorithm to be broadcast. When the requests have the finite size, the continuous-space Q-algorithm output $x$ has to be approximated by the discrete request-passing and request-dropping decisions in order to achieve the same averaged broadcast rate. When the full broadcast is expected to result in the response traffic that would be too high for the connection to handle, only the low-hop requests are supposed to be broadcast by the Q-algorithm. The high-hop requests are to be dropped. Essentially, the Q-algorithm is responsible for the GNet flow control and scalability issues when the response traffic is high – pretty much as the RR-algorithm and the OFC block are responsible for the GNet scalability when the response traffic is low.

This task is similar to the one performed by the OFC block algorithms described in section 7, which achieve the averaging goal (41) for the packet layout. So the similar algorithms could achieve the Q-algorithm averaging goals. However, it is easy to see that the algorithms described in section 7 require some buffering – in order to compare the different-hop requests, the hop-layered request buffers were introduced, and these buffers are being reset only after certain conditions are satisfied. These buffers necessarily introduce some additional latency into the GRouter data flow, and an attempt to utilize similar algorithms to achieve the Q-algorithm output averaging would also result in the additional data transfer latency for the GRouter.

Thus a different approach is suggested here. Since the fairness block algorithms already use the request buffers, it makes sense to utilize these same buffers to control the request broadcast rate according to the Q-algorithm output. This is possible since both OFC block and Q-algorithm use the same ‘hop value’ criteria to determine which requests are to be sent out and which are to be dropped. So if the ‘Q-block’ is added to the RR-algorithm, such a combined algorithm can use the same buffers to achieve the finite-message averaging for both OFC block and Q-algorithm at once. Then the Q-algorithm does not add any additional latency to the GRouter data flow, and its output just controls the Q-block of the RR-algorithm that performs the request rating, comparison and data flow averaging for both purposes.

In order to achieve that, every request arriving to the Q-algorithm is passed to the Request broadcaster (Fig. 2) – no requests are dropped by the Q-algorithm itself. However, before the request is passed to the Request broadcaster, it is assigned a ‘desired number of bytes’ (desiredBroadcastBytes) value. This is the floating-point number that tells how many bytes out of this request’s actual size the Q-algorithm would want to broadcast, if it would be possible to broadcast just a part of the request. Naturally, desiredBroadcastBytes cannot be higher than the request size (since the Q-algorithm output is limited by 100% of the incoming request traffic).

After that all the request copies are placed into the hop-layered request buffers of the other connections, so that their desiredBroadcastBytes values can be analyzed by the Q-blocks of the RR-algorithms on these
connections. The Q-block starts to work when the packet assembly is being started. It goes through the request buffers and calculates the ‘Q-volume’ for every buffer – the amount of buffer data that the Q-algorithm would want to see sent out.

The RR-algorithm and the Q-block maintain the buffer Q-volume value in a cooperative fashion. The initial buffer Q-volume value is zero. When the new request is added to the buffer, the Q-block adds the request `desiredBroadcastBytes` value to the buffer’s Q-volume. After the request buffer is sorted according to the hop-values of the requests, only the requests that are fully within the Q-volume part of the buffer are available for the RR-algorithm to be placed into the packet or to be dropped when RR-algorithm clears the request buffer. This buffer layout can be illustrated by the Fig. 11:

![Hop-layered request buffer diagram]

Fig. 11. Request buffer Q-volume and data available to the RR-algorithm.

Only the requests that fully fit within the Q-volume have a chance to be sent out (are available to the RR-algorithm). When the request is removed from the buffer by the RR-algorithm, the buffer’s Q-volume is decreased by the full size of this request. Similarly, when the multi-source herringbone stair algorithm clears the request buffer contents, it clears all the requests available to the RR-algorithm, decreasing the buffer’s Q-volume correspondingly.

Thus after the RR-algorithm resets the request buffer, the requests available to the RR-algorithm (the gray ones in Fig. 11) are going to be removed from the buffer. The resulting buffer Q-volume value will be the difference between the original Q-volume value and the size of the buffer available to the RR-algorithm:

\[
\text{Qcredit} = \text{Qvolume} - \text{bufferSizeForRR}.
\]

This remaining Q-volume value is called ‘Q-credit’, since it is used as the starting point for the Q-volume calculation when the Q-block of the RR-algorithm is invoked for the next time. It allows us to ‘average’ the discrete message-passing decisions, approximating the continuous-space Q-algorithm output over time.

Theoretically, the requests left in buffer after the RR-algorithm clears the requests available to it, (the white ones in Fig. 11) could be left in buffer and have a chance to be sent later. For example, if the first ‘white’ request in Fig. 11 (the one that has the Q-volume boundary on it) has a relatively low hop value, it could be sent out in the next OFC packet if the newly arriving requests would have the higher hop values.
In practice, however, this would result in the increased GRouter latency – such requests would spend more time in the buffer than the interval between the request buffer clearing operations. Since this is something we were trying to avoid in the first place, these requests are removed from the buffer, too - the GRouter latency minimization is considered to be more important than the better statistical layout of the data sent by the GRouter. So since we assume that the buffering requirements (intervals between buffer resets) defined by the multi-source herringbone stair algorithm (section 7.4) are enough for our purposes, we remove these requests as the buffer is cleared, too. When these requests are removed, the buffer Q-volume is not changed, so after the buffer is cleared we have an empty buffer with a Q-volume defined by the equation (48).

The Q-credit value is on the same order of magnitude as the average message size. In fact, if the Q-credit is large, the buffer Q-volume can be bigger than the whole buffer size. This does not change anything – the difference between the Q-volume and the buffer size available to RR-algorithm is still carried as the Q-credit to the next Q-block pass.

Which brings us to an interesting possibility. Let's say the very large-size request leaves a large Q-credit after the buffer is cleared, and at the same time the average request size becomes small and the incoming request traffic $f$ drops significantly – for example, this can happen when the large-message DoS attack has stopped. Then, regardless of the current Q-algorithm output, it can take us a while until we throttle down the sending operations since we are going to fully send the amount of data equal to this Q-credit value first, and act according to the Q-algorithm output ($x/f$ value) only after that.

In order to avoid that, the Q-credit left after the buffer reset is exponentially decreased over time with the characteristic time $\tau_{AV}$ equal to the characteristic time (56) (Eq. (15), [1]) of the Q-algorithm that supplies the data to this request buffer:

$$
(49) \quad \frac{dQ_{\text{credit}}}{dt} = -(1/\tau_{AV}) \times Q_{\text{credit}}.
$$

This guarantees that regardless of the instant Q-credit size due to an abnormally large request, its value will drop to 'normal' in a time comparable to the Q-algorithm characteristic time, so that the Q-algorithm would retain its traffic-controlling properties.
8.2. Response/request ratio and delay.

Q-algorithm [1] can be presented as the following set of equations:

\[
\begin{align*}
(50) \quad \frac{dQ}{dt} &= -\left(\beta\tau\tau Av\right)\left(Q - \rho B - u\right), \quad Q \leq B_{av} \\
(51) \quad u &= \max(0, Q - f R_{av}) \\
(52) \quad x &= (Q - u) / R_{av} = \min(f R_{av}, Q) / R_{av} = \min(f, Q / R_{av}) \\
(53) \quad \frac{dR_{av}}{dt} &= -(\beta\tau\tau Av)\left(R_{av} - R\right) \\
(54) \quad \frac{dB_{av}}{dt} &= -(\beta\tau\tau Av)\left(b_{av} - b\right) \\
(55) \quad \frac{dB_{av}}{dt} &= -(\beta\tau\tau Av)\left(B_{av} - B\right) \\
(56) \quad \tau\tau Av &= \max(\tau\tau R_{tt}, \tau\tau Max), \quad \text{(where } \tau\tau Max = 100 \text{ sec}) \quad \text{if } b_{av} \leq B_{av} \quad \text{and} \\
& \tau\tau Av = \tau\tau R_{tt} \quad \text{if } b_{av} > B_{av}.
\end{align*}
\]

Here the variables are:

- \(x\) -- the rate of the incoming forward-traffic (requests) passed by the Q-algorithm to be broadcast on other connections. Essentially, this variable is the Q-algorithm output.
- \(B\) -- the link bandwidth reserved for the back-traffic (responses). This variable is equivalent to \(B_i\) in terms of RR-algorithm and OFC block (section 7).
- \(\rho\) -- the part of the bandwidth \(B\) to be occupied by the average back-traffic \((\rho = 1/2)\).
- \(\beta = 1.0\) -- the negative feedback coefficient.
- \(b\) -- the actual back-traffic rate. This is the rate with which the responses to the requests \(x\) arrive from other connections. The outgoing response sending rate \(b_i\) on the connection (section 7) can be lower than \(b\), if \(b > B\) and the desired forward-traffic \(y_i\) is greater than the bandwidth reserved for the forward-traffic \(F_i\) (see Fig 4).
- \(\tau\tau Av\) -- the Q-algorithm convergence time.
- \(Q\) -- the Q-factor, which is the measure of the projected back-traffic. It is essentially the prediction of the back-traffic. The algorithm is called the 'Q-algorithm' because it controls the Q-factor for the connection. \(Q\) is limited with \(<B>\) to avoid the infinite growth of \(Q\) when \(<f R_{av}> < <\rho\) * \(B>\) and to avoid the back-stream bandwidth overflow (to maintain \(x R_{av} \leq B\) in case of the forward-traffic bursts.
- \(f\) -- the actual incoming rate of the forward traffic.
- \(R_{av}\) -- the estimated back-to-forward ratio; on the average, every byte of the requests passed through the Q-algorithm to be broadcast eventually results in \(R_{av}\) bytes of the back-traffic on that connection. This estimate is an exponentially averaged (with the same characteristic time \(\tau\tau Av\)) ratio \(R\) of actual requests and responses observed on the connection (see (53)).
- \(R\) -- the instant back-to-forward ratio; this is the ratio of actual requests and responses observed on the connection.
- \(\tau\tau R_{tt}\) -- the instant value of the response delay. This is a measure of the time that it takes for the responses to arrive for the request that is broadcast by the Q-algorithm.
- \(B_{av}\) -- the exponentially averaged value of the back-traffic link bandwidth \(B\). \((B_{av} = <B>)\)
\[ bAv \] the exponentially averaged back-traffic (response) rate \( b \), \( bAv = \langle b \rangle \)

\( u \) – the estimated underload factor. When \( u > 0 \), even if the Q-algorithm passes all the incoming forward traffic to be broadcast, it is expected that the desired part of the back-traffic bandwidth (\( \rho B \)) won’t be filled. It is introduced into the equation to limit the infinite growth of the variable \( x \) and ensure that \( x \leq f \) in that case.

The variables \( Q, u, x, Rav, Bav, bAv \) and \( tauAv \) are found from the equations (50-56), and the variables \( B, b, f, R \) and \( tauRtt \) are supplied as an input.

Furthermore, since equations (50) and (53-55) are the differential equations for the variables \( Q, Rav, bAv \) and \( Bav \) correspondingly, the system (50-56) requires the initial values for these variables. These initial values are set to zero as the calculations start. As a result, formally speaking, the equation (52) has the zero value for the \( Rav \) in the denominator on the first steps, which makes the computation of (52) impossible. In order to resolve that issue, let us notice that as the calculations are started at time \( t=0 \), the functions \( Q(t) \) and \( Rav(t) \) are going to grow as

(57) \[ Q(t) = (1/tauAv) \times (\rho \times B(t) + u(t)) \times t \quad \text{and} \]
(58) \[ Rav(t) = (1/tauAv) \times R(t) \times t \]

correspondingly when the value of \( t \) is small enough \( (t>0) \).

Since from (51) and (57) it is easy to see that \( u(t) - O(t) \), we can disregard the small \( u(t) \) in (57), which makes it clear that when \( t \) is small, the equation (52) can be written as

(59) \[ x(t) = \min(f, \rho B(t)/R(t)). \]

If \( t \) is so small that \( t < \tau Rtt \), the instant back-to-forward ratio \( R(t) \) represents just a small share of all responses for the requests issued since \( t=0 \) – all responses will take about \( R(t) \) time to arrive. So \( R(t) > 0 \) as \( t > 0 \). On the other hand, \( B(t) \) is related to the channel bandwidth and is not infinitely small when \( t > 0 \). Thus the second component in the equation (59) becomes infinitely large as \( t > 0 \), which makes it possible to write (59) and (52) as

(60) \[ x = f, \quad \text{if} \quad Rav=0. \]

That equation allows us to fully calculate the Q-algorithm output when we just start the calculations and \( Rav \) still has its initial value of \( Rav=0 \). Simply speaking, that means that when we have not seen any responses yet, we should fully broadcast all the incoming requests \( f \), since we have no way to estimate the response traffic resulting from these requests.

Now let’s have a look at the Q-algorithm input variables \( B, b, f, R \) and \( tauRtt \).
The back-traffic bandwidth $B$ ($B=B_i$, where $B_i$ is defined in Section 7) is supplied to the Q-algorithm by the RR-algorithm and OFC block (see sections 6-7, Eq. (13,14), (31) and (45,46)).

The instant traffic rates $b$ and $f$ are directly observable on the connection and can be easily measured. Note that the request traffic rate $f$ is the rate of the requests’ arrival from the Internet to the Incoming traffic-handling block in Fig. 2, whereas $b$ is the rate with which the responses arrive to the Response prioritization block from other connections.

So the missing Q-algorithm inputs are the instant response/request ratio $R$ and delay $\tau_{Rtt}$. These variables cannot be observed directly and have to be calculated from the request and response traffic streams $f$ and $b$.

In the continuous-traffic case the response traffic rate $b$ as a function of time can be presented as

$$b(t) = \int_{0}^{\infty} x(t-\tau)R(t-\tau)r(\tau)\,d\tau$$

(61)

Here $R(t)$ is the ‘true’ theoretical response/request ratio – its value determines how much response data would eventually arrive for every byte of the request broadcast $x$. The function $r(\tau)$ describes the response delay distribution over time – this normalized function (its integral from zero to infinity is equal to 1) defines the share of responses that are caused by the requests that were broadcast $\tau$ seconds ago.

Naturally, both $R(t)$ and $r(\tau)$ are not known to us and can change rapidly over time. Actually, $r(\tau)$ function in (61) should be properly written as $r(t-\tau, \tau)$ to show that the delay distribution varies over time – the first argument $t-\tau$ is omitted in (61) in order to make the physical meaning of that equation more clear.

We cannot predict the future responses, so we do not know the value of the function $R(t)$ and the shape of the function $r(\tau)=r(t, \tau)$ at any given moment $t$ – the behavior of the responses that will arrive at the future moments $t+\tau$ is not known to us. All we can do is extrapolate the past behavior of these functions. Thus we can define the Q-algorithm input $R(t)$ as:

$$R(t) = \int_{0}^{\infty} R(t-\tau)r(t-\tau, \tau)\,d\tau$$

(62)
The equation (62) describes the past behavior of the GNet in an answer to the requests and does not require any knowledge about its future behavior. All the data samples required by (62) are from the times preceding $t$, so it is always possible to calculate the instant values for $R(t)$.

The practical steps required to calculate $R(t)$ as defined in (62) are presented below.

8.2.1. Instant response/request ratio.

The instant response/request ratio $R(t)$ is defined by the equation (62). The 'true' theoretical response/request ratio $R(t)$ defines how many bytes would eventually arrive in response to every byte of requests sent out at time $t$. The 'delay function' $r(t, \tau)$ defines the delay distribution for the requests sent at time $t$; this function is normalized – its integral from zero to infinity equals $1$.

When these functions are multiplied, the result describes both how much and with what delay $\tau$ the response data arrives for the requests sent at time $t$. In the continuous traffic case this resulting response distribution function might look like the one in Fig. 12:

![Fig. 12. The response distribution over time (continuous traffic case).](image)

This sample chart shows the product of two continuous functions: the bell-shaped delay function $r(\tau) = r(t, \tau)$ and the slowly changing true return rate $R(t)$. Note that these two functions are presented separately only for the clarity – in real life we almost never can be sure that there won't be any more responses for the request sent at time $t$, so the precise separate values for $R(t)$ and for $r(t, \tau)$ can be found only postfactum, long after the request sending time $t$. $R(t) \ast r(t, \tau)$, however, has no such limitation, and as soon as the current time exceeds $t+\tau$, we have all the information needed to calculate this product on the interval $[0, \tau]$.

Essentially the equation (62) defines the latest available estimate for the response/request ratio, using the most recent responses. If we plot its integration trajectory in the same $(\tau, t)$ space that is shown in Fig. 12, it will look like a straight line with a $-45$ degree angle that starts at the current time $t$ and delay $\tau=0$:
This trajectory represents the latest available values for the $R(t-tau)^* r(t-tau, tau)$ product – the delayed responses that have arrived exactly at the moment $t$. This can be thought of as a cross-section of the plot in Fig. 12 with the vertical plane defined by the trajectory in Fig. 13.

In the real-life discrete traffic case, however, the calculation of (62) becomes more complicated. The requests and responses are not sent and received continuously as the infinitely small chunks – all networking operations are performed at the discrete time intervals and involve the finite number of bytes.

If we would plot a real-life discrete traffic response distribution in a same fashion as we did in Fig. 12, we would see a mostly zero plot of $R(t)^* r(t, tau)$ with the finite number of the infinitely high and infinitely thin peaks (delta-functions). Each such peak at the point $(tau, t)$ would represent a response that has arrived after the delay $tau$ for the request sent at time $t$. Of course, the infinitely high and infinitely thin peaks are just a convenient mathematical abstraction – their meaning is that when the packet arrives, it happens instantly from the application standpoint, so the instant receiving rate is infinite and the integral of this peak is equal to the packet size in bytes.

The sample distribution of such peaks in the same $(tau, t)$ space as in Fig. 13 is shown in Fig. 14:

Fig. 14. Sample $R(t)^* r(t, tau)$ peak distribution in $(tau, t)$ space in the discrete traffic case.
On this chart the thin horizontal lines are the 'request trajectories'. These lines start at the \( t_{\text{au}}=0 \) value when the individual requests are sent at the moment \( t \) and continue growing as the time goes on. The black marks on the request trajectories represent the individual delayed responses to these requests. The upper right corner of the chart (above the current latest response line) is empty – only the responses received so far are shown on the chart in order to simulate the realistic situation of \( R(t) \) being calculated in real time.

The plot in Fig. 14 clearly shows the difficulty of calculating \( R(t) \) in the discrete traffic case: unlike the theoretical continuous-traffic plot in Fig. 12, the integration in equation (62) has to be performed along the trajectory that typically does not have even a single non-zero value of the \( R(t-t_{\text{au}}) \) product on it. Even when the \( R(t) \) calculation is performed exactly at the moment of some response arrival, the integration trajectory still has just a few non-zero points in it, leaving most of the request trajectories (horizontal lines) outside the integration scope.

The reason for this seeming difficulty is that at any current time \( t \), the only samples of the \( R(t-t_{\text{au}}) \) product are the ones available at the moments \( t_i \), where \( t_i \) is the time when the request \( j \) has been forwarded to other connections for broadcast. At these times the value of \( R(t-t_{\text{au}}) \) is defined and available for all delay values of \( t_{\text{au}} \) not exceeding \( t-t_i \) – it is zero most of the time and is a delta-function with some weighting coefficient otherwise. However, at all other times \( t\neq t_i \) the value of the \( R(t-t_{\text{au}}) \) product is unavailable. That does not mean that it does not exist, but rather that it is not directly observable. If some request would be broadcast at that time \( t \), that fact would define the value of \( R(t-t_{\text{au}}) \) product along this request trajectory.

So the integration suggested by the plot in Fig. 14 has a logical flaw – it attempts to perform an operation (62) designed for the function that is defined everywhere on the \((t_{\text{au}}, t)\) plane, using the function that is defined only along the finite number of lines \( t=t_i \) instead. In order to perform this operation in a correct fashion we need to make the \( R(t-t_{\text{au}}) \) product value available not only at the points \((t_{\text{au}}, t)\) that correspond to the 'request trajectories', but at all other points too. Given the amount of information we have from observing the GRouter traffic, an only feasible way of achieving that is the interpolation. We have to define this function for all times \( t\neq t_i \) when it is not directly observable, using just the information from times \( t=t_i \).

In order to do that, we can act as if the requests and responses are not sent and received instantly, but gradually with finite transfer rates defined as the message sizes divided by the interval between the requests. Then the request with the size \( V_{t_i} \) is not sent instantly at the moment \( t_i \), but gradually with a finite rate \( x(t_i, t_{i+1})=V_{t_i}/(t_{i+1}-t_i) \) defined on the whole interval \([t_i, t_{i+1})\) (note that the time \( t_{i+1} \) is not included into the interval – the \( x(t_{i+1}) \) value is defined by the next request size). Thus the whole range of \( t \) is covered by these intervals and \( x(t) \) becomes non-zero everywhere. Let us use the index \( i \) to mark the responses to the individual request \( j \). Since the response \( i \) to the request \( j \) is received with the delay \( t_{\text{au}} \), this response will be also delivered gradually over the \([t_j+t_{\text{au}}, t_{j+1}+t_{\text{au}})\) interval, and if the response size is \( V_{t_{j+1}} \), the effective data transfer rate for this response will be \( V_{t_{j+1}}/(t_{j+1}-t_j) \).
This traffic-‘smoothening’ operation preserves the integral characteristics of the data transfers, and defines the \( R_t(t)^*r(t, \tau) \) product for all values of \( t \) – not only for \( t=t_0 \) allowing us to transform the plot in Fig. 14 into the one shown in Fig. 15:

![Fig. 15. \( R_t(t)^*r(t, \tau) \) value interpolation and integration in the discrete traffic case.](image)

The vertical arrows in Fig. 15 represent the non-zero values of the \( R_t(t)^*r(t, \tau) \) product and cover the interval \([t_0, t_{i+1}[^{\text{t}}\) from the request sending time \( t_i \) up to but not including the next request sending time \( t_{i+1} \). When \( t=t_{i+1} \), the new request data is used. These non-zero values are actually the delta-functions of \( \tau \) with the magnitude defined by the fact that these delta-functions are supposed to convert the request sending rate \( x(t) \) into the response receiving rate \( b(t) \) according to the equation (61).

We have already seen that the response \( i \) to the request \( j \) effectively increases the response rate on the \([t_i + \tau_u, t_{i+1} + \tau_u[\) interval by \( V_{b_j}/(t_{j+1}-t_i) \), and that this increase is caused by the request with rate \( V_{f_j}/(t_{j+1}-t_i) \) on the interval \([t_i, t_{i+1}[\). In terms of the equation (61), this additional response rate is caused by the \( R_t(t-\tau_u)^*r(t-\tau_u, \tau_u) \) product multiplied by the \( x(t-\tau_u) \) (equal to \( V_{f_j}/(t_{j+1}-t_i) \)) and by the infinitely small value \( d\tau_u \), so we can write this response rate increment as

\[
(63) \quad V_{b_j} / (t_{j+1} - t_i) = V_{f_j} / (t_{j+1} - t_i) * R_t(t-\tau_u)^*r(t-\tau_u, \tau_u) * d\tau_u,
\]

or

\[
(64) \quad V_{b_j} = V_{f_j} * R_t(t-\tau_u)^*r(t-\tau_u, \tau_u) * d\tau_u.
\]

This allows us to write the \( R_t(t)^*r(t, \tau) \) product value on the \([t_i, t_{i+1}[\) interval as

\[
(65) \quad R_t([t_i \ldots t_{i+1}[^{\text{t}}\) * \( r([t_i \ldots t_{i+1}], \tau_u) \) = \( (V_{b_j} / V_{f_j})^*\delta(\tau - \tau_u) \),
\]

where \( \delta(\tau_u-\tau_u) \) is a function which is infinite with an integral of 1 when \( \tau = \tau_u \) and zero when \( \tau \neq \tau_u \).
Equation (65) makes it possible to calculate the $R(t)$ as defined in (62) in the discrete traffic case. The continuous-space integral (62) becomes the sum, which components correspond to the non-zero points on the integration trajectory. In Fig. 15 these non-zero points can be easily seen as the vertical arrows that cross the integration trajectory. Note also that since several requests can be forwarded for broadcast at the same sending time $t_i$, this group of requests is considered a single request $j$ from the interpolation standpoint. All the replies to this group of requests are considered to be the replies to the request $j$.

However, even though this straightforward approach to the $R(t)$ computation is possible in principle, it is rather complicated in implementation and might lead to the various Q-algorithm computational errors and decreased code performance. The main problem with this integration method is that it does not take into consideration the reason for the $R(t)$ computation, which is the subsequent exponential averaging (53) and using the resulting $R_{av}$ value as the Q-algorithm input. Equation (62) allows us to calculate the value of $R(t)$ at any random moment $t$, which is first, not necessary (ultimately we need only the averaged value $R_{av}$ for the Q-algorithm), and second, results in a noisy and imprecise $R(t)$ function. In fact, it can be shown that when the time scale is discrete (as it normally is in any computer system), the integration approach illustrated in Fig. 15 leads to a systematic error proportional to the operating system ‘time quantum’ – the precision of the built-in computer clock.

The Q-algorithm equation (53) requires $R(t)$ that would correctly reflect all the response data arriving within the Q-algorithm time step $T_q$. The integration presented in Figs. 13-15 effectively counts only the very latest responses; if the Q-algorithm step time is big enough, many of the responses won’t be factored into the $R(t)$ calculation as defined in (62), which might be a source of the $R_{av}$ (and Q-algorithm) errors.

So we need $R(t)$ to be not an ‘instant’ response/request ratio at time $t$, but rather some ‘average’ value on the $[t-T_q, t]$ interval, and this ‘real-life’ $R(t)$ should be related to the Q-algorithm step size $T_q$, factoring all the responses arriving on this interval into the calculation. In order to do that, we can define the Q-algorithm input $R$ at the current time $t_k$ as $R(t_k, T_q)$, which is the average value of $R(t)$ integral (62) on the Q-algorithm step interval $[t_k-T_q, t_k]$:

$$
R(t_k, T_q) = \frac{1}{T_q} \int_{t_k-T_q}^{t_k+\infty} \int_{t_k-T_q}^{t_k+\infty} R(t, \tau)c(t-\tau, \tau) d\tau \, dt
$$

This integration approach is illustrated in Fig. 16.
Fig. 16. $R(t)*r(t, \tau)$ integration tied to the Q-algorithm step size.

Here the same response pattern as in Fig. 14 and Fig. 15 is presented together with the Q-algorithm step size $T_q$. Instead of calculating the value of $R(t)$ as suggested by Fig. 15 and equation (62), here all the responses that have the 'interpolation arrows' inside the two-dimensional integration area (shaded area in Fig. 16) are included into the equation. After the two-dimensional integral is calculated, it is divided by $T_q$ to compute $R(t, T_q)$.

It is important to realize that the integration approaches suggested in Fig. 15 (equation (62)) and Fig. 16 (equation (66)) become identical when the Q-algorithm step size $T_q>0$. We are not introducing a new definition for $R(t)$ here – we just present the discrete Q-algorithm time case approximation of the same basic function, which in the continuous Q-algorithm time case is defined by the integration along the trajectory shown in Figs. 13-15 (equation (62)). The two-dimensional integration presented in Fig. 16 is necessary because of the finite size of the Q-algorithm step time $T_q$, and not because of the discrete character of the traffic. Even if the $R(t)*r(t, \tau)$ product would be similar to the one shown in Fig. 12 and the data would be sent and received continuously in the infinitely small chunks, the two-dimensional integral (66) would still be necessary when $T_q>0$.

The discrete (finite message size) traffic, however, is the cause of the delta-function appearance in the equation (65) and of the finite-length 'interpolation arrows' in Figs. 15 and 16. So the practical computation of (66) in the discrete traffic case involves the finite number of responses – the ones that have the 'interpolation arrows' at least partly within the shaded integration area in Fig. 16. The value of every sum component is proportional to $V_{vu}/V_{fi}$ (see (65)) and to the length of the 'interpolation arrow' segment within the integration area.

Fig. 16 makes it easy to see that the response 'interpolation arrow' crosses the integration trajectory only if this response arrival time $t_i + \tau_{iu}$ is more recent than the current time $t$ minus the Q-algorithm step size $T_q$ and minus the request interval $t_{iu+1} - t_i$. So the non-zero components of the sum that replaces (66) in the discrete traffic case must satisfy the condition

$$t_i + \tau_{iu} > t - T_q - (t_{i+1} - t_i), \quad \text{or} \quad \tau_{iu} > t - T_q - t_{i+1}$$
Introducing the ‘response age’ variable $a_j = t - (t + \tau a_j)$, we can write this as:

(68)  $a_j < T_q + (t_{j+1} - t)$,  if $j$ is not the last request sent out,
(69)  $a_j \geq 0$,  if $j$ is the last request sent out (all its responses are counted).

These conditions mean that only the relatively recent responses should participate in the $R(t)$ calculation, and the maximal age of such responses should be calculated individually for every request.

Defining the length of the ‘interpolation arrow’ part that is within the integration area as $S_0 S_0(t, T_q)$ (it is written here as a function of $t$ and $T_q$ to underscore that for every response this value depends on time and on the Q-algorithm step size), from (65) and (66) we can find $R(t, T_q)$ as:

(70)  \[ R(t, T_q) = \frac{1}{T_q} \sum_{t_{j-1}}^{V_{h_j}} S_{i_j} \begin{cases} a_j < T_q + (t_{j+1} - t), & \text{if } j \text{ is not the last request} \\ a_j \geq 0, & \text{if } j \text{ is the last request} \end{cases} \]

It is not difficult to find $S_0$ at any given moment $t$, so the equation (70) can be actually implemented, giving the correct $R$ value for the Q-algorithm equation (53).

In practice, however, it is not very convenient to use the equation (70). From Fig. 16 it is clear that this sum contains not only the components related to the responses that have arrived during the last Q-algorithm step $T_q$, but also the components related to the responses received before that. So the responses’ parameters (size and arrival time) have to be stored in some lists until the corresponding response ages exceed the age limit (68). On every Q-algorithm step these lists have to be traversed to determine the old responses to be removed, then the new $S_0$ parameters have to be found for the remaining responses and only after that the sum (70) can be found.

This whole process is complicated and time-consuming, so it might be desirable to optimize it. In order to do that, let us notice that as the $T_q$ grows and the relevant ‘interpolation arrows’ have bigger chance to be fully inside the integration area, the average $S_0$ value approaches $t_{i-1} - t$. And in any case, the ‘interpolation arrow’ of every response is going to be eventually ‘fully covered’ by the integration (66) on some Q-algorithm step. Since there are no time gaps between the Q-algorithm steps, the integration areas similar to the one in Fig. 16 cover the whole $\tau a > 0$ space, and every point on every ‘arrow’ is going to belong to exactly one $S_0(t, T_q)$ interval.

Further, the equations (66) and (70) were designed to average the ‘instant’ value of $R(t)$ defined by the equation (62) over the Q-algorithm step time $T_q$, and for every two successive Q-algorithm steps $T_q1$ and $T_q2$,

(71)  \[ R(t, T_q1 + T_q2) = (R(t, T_q2) \* T_q2 + R(t - T_q2, T_q1) \* T_q1) / (T_q1 + T_q2), \]
which means that the R value for the bigger Q-algorithm step can be found as a weighted average of the R values for the smaller steps. Let us consider the model situation when there is a single response \( V_{b}\) and its 'interpolation arrow' falls into two Q-algorithm steps – \( T_q 1\) and \( T_q 2\), as shown in Fig. 17.

![Diagram showing the interpolation within two Q-algorithm steps.]

Fig. 17. Single response interpolation within two Q-algorithm steps.

Here the response 'arrow' is split into two parts \( S_i(t, T_q 2) \) and \( S_i(t-T_q 2, T_q 1) \), so

\[
(t_{i+1} - t_i) = S_i(t, T_q 2) + S_i(t-T_q 2, T_q 1).
\]

In this case the R values for these two Q-algorithm steps \( T_q 1\) and \( T_q 2\) calculated with the equation (70) are:

\[
R(t, T_q 2) = \frac{(V_{b}/ V_{f}) \cdot S_i(t, T_q 2)}{T_q 2}, \quad \text{and}
\]

\[
R(t - T_q 2, T_q 1) = \frac{(V_{b}/ V_{f}) \cdot S_i(t - T_q 2, T_q 1)}{T_q 1}.
\]

The R value for the compound step \( T_q 1+T_q 2\) is

\[
R(t, T_q 1+T_q 2) = \frac{(V_{b}/ V_{f}) \cdot (S_i(t, T_q 2) + S_i(t-T_q 2, T_q 1))}{(T_q 1 + T_q 2)}.
\]

Using (72), we can present (75) as

\[
R(t, T_q 1+T_q 2) = \frac{(V_{b}/ V_{f}) \cdot (t_{i+1} - t_i)}{(T_q 1 + T_q 2)},
\]

meaning that as the R value is being averaged over time, it does not really matter whether the response is being counted in the sum (70) precisely (according to the \( S_i\) value), or the response is just assigned to the Q-algorithm step where it was received. For example, if we simplify the R calculation and compute the R values on the two Q-algorithm steps above as:

\[
R(t, T_q 2) = 0, \quad \text{and}
\]

\[
R(t - T_q 2, T_q 1) = \frac{(V_{b}/ V_{f}) \cdot (t_{i+1} - t_i)}{T_q 1},
\]
the averaged $R$ value on these two steps will be:

\[(79) \quad R(t, Tq1+Tq2) = (Vb_y / V_f) \ast (t_{j+1} - t_j) / (Tq1 + Tq2),\]

which is identical to (76). So even though the equations (77) and (78) give us the non-precise values of the integral (66) on two individual Q-algorithm steps $Tq1$ and $Tq2$, it is a very short-term error. The averaged $R$ value on the compound interval $Tq1+Tq2$ defined by (79) is exactly the one defined by the averaging of the precise $R$ values calculated in (73) and (74).

Now, since the $R$ value is used by the Q-algorithm only as an input to the equation (53) that exponentially averages it with the characteristic time $\text{tauAv}$, we can disregard the short-term irregularities in $R$ and replace the equation (70) by the following optimized equation:

\[(80) \quad R(t, Tq) = \frac{1}{Tq} \sum_{t_j}^{t} V_i \frac{b_z}{V_f} (t_{j+1} - t_j) \quad a_y < Tq \]

Even though the equation (80) is less precise than the equation (70), its precision is sufficient for our purposes when $\text{tauAv} > t_{j+1} - t_j$. At the same time the implementation of the equation (80) is much simpler, requiring less memory and CPU cycles. Only the responses arriving within the latest Q-algorithm step time have to be counted, the complicated $S_e$ calculations do not have to be performed on every Q-algorithm step, and the memory requirements are minimal. Nothing has to be stored on ‘per response’ basis, and for every request in the routing table, just the value of the $(t_{j+1} - t_j)/V_f$ ratio has to be remembered. Then every arriving response $Vb_y$ should increase the sum in the equation (80). When the Q-algorithm step is actually done, this sum should be divided by $Tq$ to calculate $R$ and zeroed immediately after that to prepare for the next Q-algorithm step. This approach also makes it possible to ‘spread’ the calculations more evenly over the Q-algorithm time step $Tq$ instead of performing all the computations at once, as it would be the case with the equation (70).

Of course, the last request sent out should still be treated in a special way – the next request sending time $t_{j+1}$ is unavailable for it, so all its responses should be added to the sum (80) when the Q-algorithm step is actually performed. The current time $t$ should be used instead of $t_{j+1}$ in the equation (80) for this request, since $(t-t_0)/V_f$ would provide the best current estimate of the $1/x(t)$ value at this point instead of $(t_{j+1}-t_0)/V_f$ that is used as the $1/x([t_0, t_{j+1}])$ estimate for all other (previous) requests.

8.2.2. Instant delay value.

The instant delay value $\text{tauRtt}(t)$ is the measure of how long does it take for the responses to the request to arrive. The word ‘instant’ here does not imply that the responses arrive instantly – it just means that this function provides an instant ‘snapshot’ of the delays observed at the current time $t$. 

-55-
Logically this function is a weighted average value of the observed response delays $\tau$. 'Weighted' here means that the more is the amount of data in the responses with the delay $\tau$, the bigger influence should this delay value have on the value of $\tau R(t)$. This is similar to the way the instant response ratio is calculated in (62), so in principle $R(t)$ might be just replaced by $\tau$ in that equation, leading us to the following equation for $\tau R(t)$:

$$
\tau R(t) = \int_0^\infty r(t-\tau, \tau) d\tau
$$

(81)

Unfortunately the previous section (8.2.1) shows that in practice the function $r(t, \tau)$ cannot be known to us – we can never be sure that all the responses for some particular request have already arrived, and these future delayed responses might affect the past values of $r(t, \tau)$. This happens because by definition the function $r(t, \tau)$ is normalized – the integral of $r(t, \tau) d\tau$ from zero to infinity is 1. In real-life situations at any current time $t$ we do not see the full response pattern for the request $j$ sent at time $t_0$, but are limited to the requests that have arrived with the delay less or equal to $\tau = t-t_0$. The normalization requirement means that any new responses arriving after that will change the past values of $r(t, \tau)$ too, even though the responses that form this function at the values of $\tau < t-t_0$ have been already received.

Besides, the equation (81) uses the same integration trajectory as the equation (62) – the one shown in Fig. 13. So even if we would somehow know the precise values of the $r(t, \tau)$ function, the integral of $r(t-\tau, \tau) d\tau$ along this trajectory would not be equal to 1 anyway – the function $r(t, \tau)$ is normalized only for the horizontal integration trajectories $t = \text{const}$ in the $(\tau, t)$ space. Thus the direct calculation of (81) would give us the wrong value of $\tau R(t)$ when $r(t, \tau)$ changes with $t$, as it normally does.

So what we need is some practically feasible and properly normalized way to average the response delay $\tau$. This amounts to a requirement to have some function to replace $r(t-\tau, \tau)$ in (81). The solution presented uses the $R(t-\tau) r(t-\tau, \tau)$ product for this purpose.

As an averaging multiplier for $\tau$, this function has some very attractive properties: first, its calculation does not require any knowledge about the future data, which means that the future responses won’t change the values that we already have.

Second, this function is pretty close to the $r(t-\tau, \tau)$, differing only by the true response/request ratio value $R(t)$, and it can be argued that this multiplier actually makes sense from the averaging standpoint. For example, the requests with many responses would have stronger influence on the $\tau R(t)$, meaning that generally $\tau R(t)$ would be closer to the average response time for the requests that provide the bulk of the return traffic.
Third, as long as the function used for the \( \tau \) averaging instead of \( r(t-\tau, \tau) \) in (81) has some defensible relationship to the response distribution pattern \( r(t-\tau, \tau) \) (as \( R(t-\tau) r(t-\tau, \tau) \) product certainly does), it is a matter of the secondary importance, which particular function is used. The \( \tau_{\text{Rt}}(t) \) variations due to the different averaging function choice can be countered by the appropriate choice of the negative feedback coefficient \( \beta \) for the equations (50) and (53-55), since the value of \( \tau_{\text{Rt}} \) just controls the Q-algorithm convergence rate and does not affect anything else. In fact, even that function of \( \tau_{\text{Rt}} \) is present only when the response bursts with rate \( b>B \) are observed. Normally, when there’s no response burst and \( \tau_{\text{Rt}} \) is not very big (\( \tau_{\text{Rt}}<\tau_{\text{Max}} \)), the Q-algorithm convergence speed is limited by the bigger time \( \tau_{\text{Max}} \) anyway, as defined by (56). In practice, being close to \( r(t-\tau, \tau) \), our particular averaging function choice does not require changing \( \beta \) from its recommended value of 1.0.

And finally, we are calculating the values related to the \( \mathbf{R}(t-\tau) r(t-\tau, \tau) \) product and its integral anyway when we are calculating \( \mathbf{R}(t) \) as described in section 8.2.1.

The only unattractive property of \( \mathbf{R}(t-\tau) r(t-\tau, \tau) \) product as an averaging function is that its integral is not normalized to 1 over the integration trajectory shown in Fig. 13. However, this is easily fixed by explicitly normalizing this product by dividing it by \( \mathbf{R}(t) \), which is exactly the value of this integral (62) over the integration trajectory in Fig 13.

So we can present the expression for \( \tau_{\text{Rt}}(t) \) as:

\[
\tau_{\text{Rt}}(t) = \frac{1}{\mathbf{R}(t)} \int_{0}^{+\infty} \tau \cdot \mathbf{R}(t-\tau) r(t-\tau, \tau) d\tau
\]

(82)

Applying the same line of reasoning as the one applied in section 8.2.1 to the similar equation (62), in the discrete traffic case we can replace (82) by a finite sum

\[
\tau_{\text{Rt}}(t) = \frac{1}{\mathbf{R}(t)} \sum_{i,j} \frac{a_{ij} T_{q}}{V_{ij} V_{qi}} (t_{ij+1} - t_{ij}) \quad \text{if} \quad a_{ij} < T_{q}
\]

(83)

in the same fashion as we have replaced (62) by its discrete-traffic representation (80). Here the sum components are calculated in a fashion similar to (80) – in fact, both sums (80) and (83) can be calculated in parallel as the responses arrive, and then the value of \( \mathbf{R}(t) \) from (80) can be used to normalize the sum in (83) to calculate the \( \tau_{\text{Rt}}(t) \) value.
The same last request treatment rules that were described in section 8.2.1 for the equation (80) apply to
the equation (83). All responses to this request should be included into the sum (83) and the current time t
should be used instead of the next request sending time t_{0+1}.

Naturally, the equation (83) is inapplicable when R(t)=0. Consider the case when on the average
there's less than one response per request j (actually, request group j). This situation is particularly likely to
arise when the number of requests in the average request group j is small. Then on the average there's likely to
be no non-zero response components in (80) and (83), meaning that both R(t) and the sum in (83) would be
equal to zero. In that case the previous value of tauRtt should be used. If no previous tauRtt values are
available, that means that the connection was just opened and no requests forwarded by it for broadcast to other
connections have resulted in the responses yet. Then we cannot estimate R(t) and tauRtt(t), so the initial
conditions described in Section 8.2 (equation (60)) should apply to x(t) and tauRtt=0 should be used in (56).

When tauRtt(t) is calculated on the basis of just a few data samples (or even a single data sample), the
value of tauRtt(t) might have a big variance. Of course, the same would be also true for the R(t) function, but
that function is used by the Q-algorithm only after the averaging over the tauAv time period (equation (53)).
The tauRtt(t), on the contrary, is used directly in (56), since it is this value that might be defining the averaging
interval for all other equations ((50) and (53-55)), and it might be difficult to average it exponentially in a
similar fashion.

Fortunately the value of tauRtt is used only when the long response traffic burst is present or when
tauRtt>tauMax (56). Otherwise, the constant value tauMax (56) defines the Q-algorithm convergence rate, so
normally tauRtt is not used by the Q-algorithm at all. But even when it is used by the Q-algorithm, it just
defines the algorithm convergence speed and if the general numerical integration guidelines presented in
Appendix B are observed, the big tauRtt variance should not present a problem.

However, the extremely high variance of tauRtt is still undesirable, so it is recommended to calculate
tauRtt on the basis of at least 10 response samples or so, increasing the Tq averaging interval in the equation
(83) if necessary. This is made even more important by the fact that the equation (83) is the analog of the
optimized approximation (80) for R(t) and not of the precise equation (70), which might lead to the higher
variance of tauRtt because of this approximate computation. Thus the bigger averaging interval Tq might be
desirable, so that the average interval t_{0+1} between requests would be less than Tq, since t_{0+1}-t_t<<Tq is the
condition required for the approximate solution (80) to converge to the precise solution (70).

Finally it should be noted that the interaction between the Q-algorithm and the RR-algorithm and OFC
block described in section 8.1 makes it very difficult to determine whether the individual request was sent out or
not. This information would have to be communicated in a complicated fashion from the RR-algorithms of
several connection blocks to the Q-algorithm of the connection block that has received the request. In principle
it is possible to do so, however, it is much simpler to consider every request passing through the Q-algorithm
'partially broadcast' with the request size equal to
\[(84) \quad V_{ef} = V_{req} \cdot (x(t)/f(t)),\]

where \(V_{req}\) is the actual request message size, \(x(t)/f(t)\) is the Q-algorithm output and \(V_{ef}\) is the resulting effective request size. The \(V_{ef}\) value to be used in the equations (80) and (83) is defined as:

\[(85) \quad V_{ef} = \sum(V_{ef}) \quad \text{for all the requests forwarded on the current Q-algorithm step.}\]

The effective request size \(V_{ef}\) is essentially the ‘desired number of bytes’ to be broadcast from this request as defined in section 8.1 – that’s how many request bytes the Q-algorithm would wish to broadcast if it would be possible to broadcast just a part of the request. This value is associated with the request when it is passed to the OFC block. \(V_{ef}\) is the summary desired number of bytes to send on the current Q-algorithm step. This value (or the related \((t_{b-1:t})/V_{ef}\) value) is associated with every request in the routing table and is used in the equations (80) and (83).

Since the actual requests are atomic and can be either sent or discarded, this fact also increases the variance of \(R(t)\) and \(\tau_{Rtt}(t)\). For example, all the requests forwarded for broadcast on some Q-algorithm step can be actually dropped and thus have no responses, which would result in the zero response traffic caused by the forward data transfer rate \(x(t)\) on this Q-algorithm step. And all the requests forwarded on the next Q-algorithm step might be sent out and cause the response traffic that would be disproportional for this step’s \(x(t)\).

This underscores the need to compute \(\tau_{Rtt}(t)\) only when many (much more than one) response data samples are available for the equation (83). Unlike \(R(t)\) that is averaged by \((53)\), \(\tau_{Rtt}(t)\) is being averaged only by the equation (83) itself, and the additional variance arising from the atomic nature of the requests has to be suppressed when \(\tau_{Rtt}\) is computed.

**9. Recapitulation of Selected Embodiments.**

This section briefly highlights and identifies and recapitulates particular embodiments of algorithms and architectural decisions introduced in the previous sections. These selections are by way of example and not limitation.

Section 3: The Gnmtella router (GRouter) block diagram is introduced. The ‘Connection 0’, or the ‘virtual connection’ is presented as the API to the local request-processing block (see Appendix A for the details).

Section 4: The Connection Block diagram is introduced and the basic message processing flow is described.

Section 6.1: The algorithm to determine the desirable network packet size to send is presented (equations (2-4)).
Section 6.2: The algorithms used to determine when the packet has to be sent (G-Nagle and wait time algorithm – equations (9-11)) are described. The algorithm to determine the outgoing bandwidth estimate (equations (13,14)) is presented.

Section 7.1: The simplified bandwidth layout (equations (25,26)) is introduced.

Section 7.2: The method to satisfy the bandwidth reservation requirement by varying the packet layout (equations (39,40)) is presented.

Section 7.3: The ‘herringbone stair’ algorithm is introduced. This algorithm satisfies the bandwidth reservation requirements in the discrete traffic case. The equations (45) and (46) are introduced to determine the outgoing response bandwidth estimate.

Section 7.4: The ‘herringbone stair’ algorithm is extended to handle the situation of multiple incoming data streams.

Section 8.1: The Q-block of the RR-algorithm is introduced. The goal of this block is to provide the interaction between the Q-algorithm and the RR-algorithm in order to minimize the Q-algorithm latency.

Section 8.2: The initial conditions for the Q-algorithm are introduced, including the case of the partially undefined Q-algorithm input (equation (60)).

Section 8.2.1: The algorithm to compute the instant response/request ratio for the Q-algorithm is described (equations (68-70)). The optimized method to compute the same value is proposed (equation (80)).

Section 8.2.2: The algorithm for the instant delay value computation (equation (83)) is presented. The methods to compute the effective request size for the OFC block and for the equations (80), (83) are introduced (equations (84) and (85)).

The foregoing descriptions of specific embodiments of the present invention have been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise forms disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the claims appended hereto and their equivalents. All publications and patent applications cited in this specification are herein incorporated by reference as if each individual publication or patent application were specifically and individually indicated to be incorporated by reference.
10. References.

The Gnutella protocol

Last update: 2 July 2000

Help shape the next version of Gnutella genehkan@xcf.berkeley.edu

Notes
Everything is in network byte order unless otherwise noted. Byte order of the GUID is not important.

Apparently, there is some confusion as to what "\r" and "\n" are. Well, \r is carriage return, or 0x0d, and \n is newline, or 0x0a. This is standard ASCII, but there it is, from "man ascii".

Keep in mind that every message you send can be replied by multiple hosts. Hence, Ping is used to discover hosts, as the Pong (Ping reply) contains host information.

Throughout this document, the term server and client is interchangeable. Gnutella clients are Gnutella servers.

Thanks to capnbry for his efforts in decoding the protocol and posting it.

How GnutellaNet works
General description

GnutellaNet works by "viral propagation". I send a message to you, and you send it to all clients connected to you. That way, I only need to know about you to know about the entire rest of the network.

A simple glance at this message delivery mechanism will tell you that it generates inordinate amounts of traffic. Take for example the defaults for Gnutella 0.54. It defaults to maintaining 25 active connections with a TTL (TTL means Time To Live, or the number of times a message can be passed on before it "dies"). In the worst of worlds, this means 25^7 or 6103515625 (6 billion) messages resulting from just one message!
Well, okay. In truth it isn't that bad. In reality, there are less than two thousand Gnutella clients on the GnutellaNet at any one time. That means that long before the TTL expires on our hypothetical message, every client on the GnutellaNet will have seen our message.

Obviously, once a client sees a message, it's unnecessary for it to process the message again. The original Gnutella designers, in recognition of this, engineered each message to contain a GUID (Globally Unique Identifier) which allows Gnutella clients to uniquely identify each message on the network.

So how do Gnutella clients take advantage of the GUID? Each Gnutella client maintains a short memory of the GUIDs it has seen. For example, I will remember each message I have received. I forward each message I receive as appropriate, unless I have already seen the message. If I have seen the message, that means I have already forwarded it, so everyone I forwarded it to has already seen it, and so on. So I just forget about the duplicate and save everyone the trouble.

**Topology**

The GnutellaNet has no hierarchy. Every server is equal. Every server is also a client. So everyone contributes. Well, as in all egalitarian systems, some servers are more equal than others. Servers running on fast connections can support more traffic. They become a hub for others, and therefore get their requests answered much more quickly. Servers on slow connections are relegated to the backwaters of the GnutellaNet, and get search results much more slowly. And if they pretend to be fast, they get flooded to death.

But there's more to it than that.

Each Gnutella server only knows about the servers that it is directly connected to. All other servers are invisible, unless they announce themselves by answering to a PING or by replying to a QUERY. This provides amazing anonymity.

Unfortunately, the combination of having no hierarchy and the lack of a definitive source for a server list means that the network is not easily described. It is not a tree (since there is no hierarchy) and it is cyclic. Being cyclic means there is a lot of needless network traffic. Clients today do not do much to reduce the traffic, but for the GnutellaNet to scale, developers will need to start thinking about that.

**Connecting to a server**

After making the initial connection to the server, you must handshake. Currently, the handshake is very simple. The connecting client says:

```
GNUTELLA CONNECT/0.4
```

The accepting server responds:

```
GNUTELLA OK
```

.../wego.pages.page?groupId=116705&view=page&pageId=119598&folderId=116767&panelId4/3/2001
After that, it's all data.

**Downloading from a server**

Downloading files from a server is extremely easy. It's HTTP. The downloading client requests the file in the normal way:

```
GET /get/1234/strawberry-rhubarb-pies.rcp HTTP/1.0\r\nConnection: Keep-Alive\r\nRange: bytes=0-\r\n\r\n```

As you can see, Gnutella supports the range parameter for resuming partial downloads. The 1234 is the file index (see HITS section, below), and "strawberry-rhubarb-pies.rcp" is the filename.

The server will respond with normal HTTP headers. For example:

```
HTTP 200 OK\r\nServer: Gnutella\r\nContent-type: application/binary\r\nContent-length: 948\r\n\r\n```

The important bit is the "Content-Length" header. That tells you how much data to expect. After you get your fill, close the socket.

**Header**

<table>
<thead>
<tr>
<th>bytes summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15 Message identifier</td>
<td>This is a Windows GUID. I'm not really sure how globally-unique this has to be. It is used to determine if a particular message has already been seen.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Ping</td>
</tr>
<tr>
<td>0x01</td>
<td>Pong (Ping reply)</td>
</tr>
<tr>
<td>0x40</td>
<td>Push request</td>
</tr>
<tr>
<td>0x80</td>
<td>Query</td>
</tr>
<tr>
<td>0x81</td>
<td>Query hits (Query reply)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>17 TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to live. Each time a message is forwarded its TTL is decremented by one. If a message is received with TTL of less than one (1), it should not be forwarded.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18 Hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of times this message has been forwarded.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19-22 Payload length</th>
</tr>
</thead>
<tbody>
<tr>
<td>The length of the ensuing payload.</td>
</tr>
</tbody>
</table>

**Payload: ping (function 0x00)**

**No payload**

**Routing instructions**

```
.../wego.pages.page?groupId=116705&view=page&parent=119598&folderId=116767&panelId=4/3/2001
```
Forward PING packets to all connected clients. Most other documents state that you should not forward packets to their originators. I think that's a good optimization, but not a real requirement. A server should be smart enough to know not to forward a packet that it originated.

A cursory analysis of GnutellaNet traffic shows that PING comprises roughly 50% of the network traffic. Clearly, this needs to be optimized. One of the problems with clients today is that they seem to PING the network periodically. That is indeed necessary, but the frequency of these "update" PINGs can be drastically reduced. Simply watching the PONG messages that your client routes is enough to capture lots of hosts.

One possible way to really reduce the number of PINGs is to alter the protocol to support PING messages which includes PONG data. That way you need only wait for hosts to announce themselves, rather than discovering them yourself.

Payload: pong (query reply) (function 0x01)

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>Port</td>
<td>IPv4 port number.</td>
</tr>
<tr>
<td>2-5</td>
<td>IP address</td>
<td>IPv4 address. <strong>x86 byte order! Little endian!</strong></td>
</tr>
<tr>
<td>6-9</td>
<td>Number of files</td>
<td>Number of files the host is sharing.</td>
</tr>
<tr>
<td>10-13</td>
<td>Number of kilobytes</td>
<td>Number of kilobytes the host is sharing.</td>
</tr>
</tbody>
</table>

Routing instructions

Like all replies, PONG packets are "routed". In other words, you need to forward this packet only back down the path its PING came from. If you didn't see its PING, then you have an interesting situation that should never arise. Why? If you didn't see the PING that corresponds with this PONG, then the server sending this PONG routed it incorrectly.

Payload: query (function 0x80)

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>Minimum speed</td>
<td>The minimum speed, in kilobytes/sec, of hosts which should reply to this request.</td>
</tr>
<tr>
<td>2+</td>
<td>Search criteria</td>
<td>Search keywords or other criteria. NULL terminated.</td>
</tr>
</tbody>
</table>

Routing instructions

Forward QUERY messages to all connected servers.

Payload: query hits (query reply) (function 0x81)

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 N</td>
<td>Number of hits</td>
<td>The number of hits in this set. See &quot;Result set&quot; below.</td>
</tr>
<tr>
<td>1-2</td>
<td>Port</td>
<td>IPv4 port number.</td>
</tr>
<tr>
<td>3-6</td>
<td>IP address</td>
<td>IPv4 address. <strong>x86 byte order! Little endian!</strong></td>
</tr>
</tbody>
</table>
7-10 Speed
11+ Result set

Speed, in kilobits/sec, of the responding host.
There are N of these (see "Number of hits" above).

**bytes summary description**

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>Index</td>
<td>Index number of file.</td>
</tr>
<tr>
<td>4-7</td>
<td>Size</td>
<td>Size of file in bytes.</td>
</tr>
<tr>
<td>8+</td>
<td>File name</td>
<td>Name of file. Terminated by double-NUL.</td>
</tr>
</tbody>
</table>

Last 16 Client identifier bytes

GUID of the responding host. Used in PUSH.

**Routing instructions**

HITS are routed. Send these messages back on their inbound path.

**Payload: push request (function 0x40)**

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15</td>
<td>Client identifier</td>
<td>GUID of the host which should push.</td>
</tr>
<tr>
<td>16-19</td>
<td>Index</td>
<td>Index number of file (given in query hit).</td>
</tr>
<tr>
<td>20-23</td>
<td>IP address</td>
<td>IPv4 address to push to.</td>
</tr>
<tr>
<td>24-25</td>
<td>Port</td>
<td>IPv4 port number to push to.</td>
</tr>
</tbody>
</table>

**Routing instructions**

Forward PUSH messages only along the path on which the query hit was delivered. If you missed the query hit then drop the packet, since you are not instrumental in the delivery of the PUSH request.

**Need some feedback on this one.**

**Developers' Corner**

Chat with the people who make Gnutella happen in the #gnutella and #gnutelladev channels on Efnet IRC. The Gnutella Working Group is open at http://gnutelladev.wego.com. All developers should join and participate so we can all know what we all need to do to make GnutellaNet continue to thrive. And, if you're interested in working on the Next Generation Gnutella Protocol, join the GnutellaNG Working Group at http://gnutellaneg.wego.com.

Sign up for the Gnutella Developers Mailing List

Copyright © 1999-2001 WEGO SYSTEMS, Inc. All Rights Reserved. Wego.com, Inc., does not claim ownership of, or take, or have responsibility for any content on this site.
Gen Kan's Document of the Gnutella Protocol

Notes:

Everything is in network byte order unless otherwise noted. Byte order of the GUID is not important.

Apparently, there is some confusion as to what "\r" and "\n" are. Well, \r is carriage return, or 0x0d, and \n is newline, or 0x0a. This is standard ASCII, but there it is, from "man ascii".

Keep in mind that every message you send can be replied by multiple hosts. Hence, Ping is used to discover hosts, as the Pong (Ping reply) contains host information.

Throughout this document, the term server and client is interchangeable. Gnutella clients are Gnutella servers.

Thanks to capnbry for his efforts in decoding the protocol and posting it.

How GnutellaNet Works:

General description:

GnutellaNet works by "viral propagation". I send a message to you, and you send it to all clients connected to you. That way, I only need to know about you to know about the entire rest of the network.

A simple glance at this message delivery mechanism will tell you that it generates inordinate amounts of traffic. Take for example the defaults for Gnutella 0.54. It defaults to maintaining 25 active connections with a TTL (TTL means Time To Live, or the number of times a message can be passed on before it "dies"). In the worst of worlds, this means $25^{7}$, or 6103515625 (6 billion) messages resulting from just one message!

Well, okay. In truth it isn't that bad. In reality, there are less than two thousand Gnutella clients on the GnutellaNet at any one time. That means that long before the TTL expires on our hypothetical message, every client on the GnutellaNet will have seen our message.

Obviously, once a client sees a message, it's unnecessary for it to process the message again. The original Gnutella designers, in recognition of this, engineered each message to contain a GUID (Globally Unique Identifier) which allows Gnutella clients to uniquely identify each message on the network.

So how do Gnutella clients take advantage of the GUID? Each Gnutella client maintains a short memory of the GUIDs it has seen. For example, I will remember each message I have received. I forward each message I receive as appropriate, unless I have already seen the message. If I have seen the message, that means I have already forwarded it, so everyone I forwarded it to has already seen it, and so on. So I just forget about the duplicate and save everyone the trouble.

Topology:


4/3/2001
The GnutellaNet has no hierarchy. Every server is equal. Every server is also a client. So everyone contributes. Well, as in all egalitarian systems, some servers are more equal than others. Servers running on fast connections can support more traffic. They become a hub for others, and therefore get their requests answered much more quickly. Servers on slow connections are relegated to the backwaters of the GnutellaNet, and get search results much more slowly. And if they pretend to be fast, they get flooded to death.

But there's more to it than that.

Each Gnutella server only knows about the servers that it is directly connected to. All other servers are invisible, unless they announce themselves by answering to a PING or by replying to a QUERY. This provides amazing anonymity.

Unfortunately, the combination of having no hierarchy and the lack of a definitive source for a server list means that the network is not easily described. It is not a tree (since there is no hierarchy) and it is cyclic. Being cyclic means there is a lot of needless network traffic. Clients today do not do much to reduce the traffic, but for the GnutellaNet to scale, developers will need to start thinking about that.

Connecting to a server
After making the initial connection to the server, you must handshake. Currently, the handshake is very simple. The connecting client says:

GNUTELLA CONNECT/0.4

The accepting server responds:

GNUTELLA OK

After that, it's all data.

Downloading from a server
Downloading files from a server is extremely easy. It's HTTP. The downloading client requests the file in the normal way:

GET /get/1234/strawberry-rhubarb-pies.rcp HTTP/1.0
Connection: Keep-Alive
Range: bytes=0-\n
As you can see, Gnutella supports the range parameter for resuming partial downloads. The 1234 is the file index (see HITS section, below), and "strawberry-rhubarb-pies.rcp" is the filename.

The server will respond with normal HTTP headers. For example:

HTTP 200 OK
Server: Gnutella
Content-type: application/binary
Content-length: 948

The important bit is the "Content-Length" header. That tells you how much data to expect. After you

get your fill, close the socket.

Header

<table>
<thead>
<tr>
<th>bytes summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15 Message identifier</td>
<td>This is a Windows GUID. I'm not really sure how globally-unique this has to be. It is used to determine if a particular message has already been seen.</td>
</tr>
<tr>
<td>16 Payload descriptor (function identifier)</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>0x00 Ping</td>
</tr>
<tr>
<td></td>
<td>0x01 Pong (Ping reply)</td>
</tr>
<tr>
<td></td>
<td>0x40 Push request</td>
</tr>
<tr>
<td></td>
<td>0x80 Query</td>
</tr>
<tr>
<td></td>
<td>0x81 Query hits (Query reply)</td>
</tr>
<tr>
<td>17 TTL</td>
<td>Time to live. Each time a message is forwarded its TTL is decremented by one. If a message is received with TTL less than one (1), it should not be forwarded.</td>
</tr>
<tr>
<td>18 Hops</td>
<td>Number of times this message has been forwarded.</td>
</tr>
<tr>
<td>19-22 Payload length</td>
<td>The length of the ensuing payload.</td>
</tr>
</tbody>
</table>

Payload: ping (function 0x00)

No payload

Routing instructions

Forward PING packets to all connected clients. Most other documents state that you should not forward packets to their originators. I think that's a good optimization, but not a real requirement. A server should be smart enough to know not to forward a packet that it originated.

A cursory analysis of GnutellaNet traffic shows that PING comprises roughly 50% of the network traffic. Clearly, this needs to be optimized. One of the problems with clients today is that they seem to PING the network periodically. That is indeed necessary, but the frequency of these "update" PINGs can be drastically reduced. Simply watching the PONG messages that your client routes is enough to capture lots of hosts.

One possible way to really reduce the number of PINGs is to alter the protocol to support PING messages which includes PONG data. That way you need only wait for hosts to announce themselves, rather than discovering them yourself.

Payload: pong (query reply) (function 0x01)

<table>
<thead>
<tr>
<th>bytes summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1 Port</td>
<td>IPv4 port number.</td>
</tr>
<tr>
<td>2-5 IP address</td>
<td>IPv4 address. x86 byte order! Little endian!</td>
</tr>
<tr>
<td>6-9 Number of files</td>
<td>Number of files the host is sharing.</td>
</tr>
<tr>
<td>10-13 Number of kilobytes</td>
<td>Number of kilobytes the host is sharing.</td>
</tr>
</tbody>
</table>


4/3/2001
Routing instructions
Like all replies, PONG packets are "routed". In other words, you need to forward this packet only back down the path its PING came from. If you didn't see its PING, then you have an interesting situation that should never arise. Why? If you didn't see the PING that corresponds with this PONG, then the server sending this PONG routed it incorrectly.

Payload: query (function 0x80)

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>Minimum speed</td>
<td>The minimum speed, in kilobytes/sec, of hosts which should reply to this request.</td>
</tr>
<tr>
<td>2+</td>
<td>Search criteria</td>
<td>Search keywords or other criteria. NULL terminated.</td>
</tr>
</tbody>
</table>

Routing instructions
Forward QUERY messages to all connected servers.

Payload: query hits (query reply) (function 0x81)

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Number of hits (N)</td>
<td>The number of hits in this set. See &quot;Result set&quot; below.</td>
</tr>
<tr>
<td>1-2</td>
<td>Port</td>
<td>IPv4 port number.</td>
</tr>
<tr>
<td>3-6</td>
<td>IP address</td>
<td>IPv4 address. x86 byte order! Little endian!</td>
</tr>
<tr>
<td>7-10</td>
<td>Speed</td>
<td>Speed, in kilobits/sec, of the responding host.</td>
</tr>
<tr>
<td>11+</td>
<td>Result set</td>
<td>There are N of these (see &quot;Number of hits&quot; above).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>Index</td>
<td>Index number of file.</td>
</tr>
<tr>
<td>4-7</td>
<td>Size</td>
<td>Size of file in bytes.</td>
</tr>
<tr>
<td>8+</td>
<td>File name</td>
<td>Name of file. Terminated by double-NULL.</td>
</tr>
</tbody>
</table>

Last 16 bytes Client identifier GUID of the responding host. Used in PUSH.

Routing instructions
HITS are routed. Send these messages back on their inbound path.

Payload: push request (function 0x40)

<table>
<thead>
<tr>
<th>bytes</th>
<th>summary</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15</td>
<td>Client identifier</td>
<td>GUID of the host which should push.</td>
</tr>
<tr>
<td>16-19</td>
<td>Index</td>
<td>Index number of file (given in query hit).</td>
</tr>
<tr>
<td>20-23</td>
<td>IP address</td>
<td>IPv4 address to push to.</td>
</tr>
<tr>
<td>24-25</td>
<td>Port</td>
<td>IPv4 port number to push to.</td>
</tr>
</tbody>
</table>

Routing instructions
Forward PUSH messages only along the path on which the query hit was delivered. If you missed the query hit then drop the packet, since you are not instrumental in the delivery of the PUSH request.

Need some feedback on this one.
Document of the Gnutella Protocol

A Non-technical Introduction:

The Gnutella network is is a form of a distributed file sharing system. That is, each host connected to the network is in theory considered equal. In pseudo-distributed file sharing systems such as Napster or Scour Exchange, each client connects to one or more central servers. With gnutellaNet, there is no centralized server. Each client also functions as a server. This way, the network becomes much more immune to shutdown or regulation.

Technical Theory of Operation:

The Gnutella network is a collection of Gnutella servants that cooperate in maintaining the network.

A broadcast packet on the Gnutella network begins its life at a single servant, and is broadcasted to all connected servants. These servants then rebroadcast to all connected servants. This continues until the time to live of the packet expires.

If all servants have eight other servants connected, one broadcast packet with a time to live of 7 can make it to 8^7 servants, which is 2097152. This is far more than enough to reach all the servants on the Gnutella network.

A reply packet on the Gnutella network begins its life as a response to a broadcast packet. It is forwarded back to the servant where its initiating broadcast came from until it gets back to the servant that sent off the broadcast.

To keep track of where a packet came from, each packet is prefixed with a 16 byte Message ID. The Message ID is simply random data. A servant uses the same Message ID for all of its own broadcast packets.

Each servant keeps a hash table of the most recent few thousand packets it has received. The hash table matches the Message ID with the IP address the message came from.

To route a reply packet back where it came from, a servant checks its hash table for the Message ID, and sends it back to the IP address the Message ID is matched to. This continues until the packet gets back home.

This results in a network with no hierarchy, every servant is equal. In order to be part of the network, one must contribute to the network. However, some servants are more equal than others. Servants running on faster internet connections are more suited to hub (maintain more GnutellaNet connections) than others, and therefore get responses from the network much faster.

Each Gnutella server only knows about the servers that it is directly connected to. All other servers are invisible, unless they announce themselves by answering to a PING or by replying to a QUERY. This provides amazing anonymity.

Unfortunately, the combination of having no hierarchy and the lack of a definitive source for a server list means that the network is not easily described. It is not a tree (since there is no hierarchy) and it is

cyclic. Being cyclic means there is a lot of needless network traffic. Clients today don't do much to reduce the traffic, but for the GnutellaNet to scale, developers will need to start thinking about that.

Connecting:

The initiator opens a TCP connection. The initiator sends 'GNUTELLA CONNECT/0.4\n\n'. The receiver sends 'GNUTELLA OK\n\n'. After this, it's all packets.

Header:

Bytes 0 - 15: Message ID:
A Message ID is generated on the client for each new message it creates. The Message ID is 16 bytes of random data.

Byte 16: Function ID:
What message type the packet is. See the list of function types below for descriptions of the types.

Byte 17: TTL Remaining:
How many hops the packet has left before it should be dropped.

Byte 18: Hops taken:
How many hops this packet has already taken. Set the TTL on response messages to this value!

Bytes 19 - 22: Data Length:
The length of the Function-dependant data which follows. There has been some discussion as to if this value is actually only 2 bytes and the last 2 bytes are something else. Seems to work with 4 for me. Also there is a question as to signed or unsigned integers. Don't know that either, I can't get gnutella to try and send a 2^31 + 1 byte packet :).

List of Functions:

- 0: Ping
- 1: Ping Reply (Pong)
- 64: Push Request
- 128: Query
- 129: Query Reply (Hits)

Ping:

A Ping has no body.

Routing:
Rebroadcast packet through every available connection, except the one it was received from.

Ping Reply (Pong):

Bytes 0 - 1: Host port:
The TCP port number of the listening host

http://www.gnutelladev.com/docs/our-protocol.html
Bytes 2 - 5: Host IP:
The IP address of the listening host, in network byte order.

Bytes 6 - 9: File Count:
An integer value indicating the number of files shared by the host. No idea if this is a signed or unsigned value.

Bytes 10 - 14: Files Total Size
An integer value indicating the total size of files shared by the host, in kilobytes (KB). No idea if this is a signed or unsigned value.

Routing:
Forward packet only through the connection that the Ping came from.

Query:

Bytes 0 - 1: Minimum Speed:
The minimum speed of serverants which should perform the search. This is entered by the user in the "Minimum connection speed" edit box.

Bytes 2+: Search String:
A NULL terminated character string which contains the search request. Routing:
Rebroadcast packet through every available connection, except the one it was received from.

Query Reply (Hits):

Byte 0: Number of Items:
Number of Search Reply Items (see below) which follow this header.

Bytes 1 - 2: Host Port:
The listening port number of the host which found the results.

Bytes 3 - 6: Host IP:
The IP address of the host which found the results. In network byte order.

Bytes 7 - 9: Host Speed:
The speed of the host which found the results.

Bytes 10+: List of Items:
A Search Response Item (see below) for each result found.

Last 16 Bytes: Footer:
The clientID128 of the host which found the results. This value is stored in the gnutella.ini and is a GUID created with CoCreateGUID() the first time gnutella is started.

Routing:
Forward packet only through the Query came from.

http://www.gnutelladev.com/docs/our-protocol.html

4/3/2001
Push Request (Query Reply Reply):

*Bytes 0 - 15: ClientID128:*
The ClientID128 GUID of the server the client wishes the push from.

*Bytes 16 - 19: File Index:*
Index of file requested. See Search Reply Items for more info.

*Bytes 20 - 23: Requester IP:*
IP Address of the host requesting the push. Network byte order.

*Bytes 24 - 25: Requester Port:*
Port number of the host requesting the push.

Search Reply Items:

*Bytes 0 - 3: File Index:*
Each file indexed on the server has an integer value associated with it. When Gnutella scans the hard drive on the server a sequential number is given to each file as it is found. This is the file index.

*Bytes 4 - 7: File Size:*
The size of the file (in bytes).

*Bytes 8+: File Name:*
The name of the file found. No path information is sent, just the file's name. The filename field is double-NUL terminated.

Downloading:

Downloading is done by HTTP. A GET request is sent, with a URI that is constructed from the information in a Search Reply. The URI starts with /get/, then the File Index number (see Search Reply Items), then the filename. Example download request:

```
GET /get/1234/strawberry-rhubarb-pies.rsp HTTP/1.0\nConnection: Keep-Alive\nRange: bytes=0-\n\n```

The server should respond with normal HTTP headers, then the file.

```
HTTP 200 OK\nServer: Gnutella\nContent-type: application/binary\nContent-length: 948\n\n```

Uploading:

Uploading is done in response to a Push Request. The uploader establishes a TCP connection, and sends GIV, then the File Index number, then the ClientID128 of the uploader, and then the filename.

Example:

GIV 1234/abcdefghijklmnop/Strawberry_Rhubarb_Pie.txt

Home: Docs

http://www.gnutelladev.com/docs/our-protocol.html
Cap'n Bry's Document of the Gnutella Protocol

I've started work on a program which utilizes the gnutella protocol (as of version 0.48). Basically, you connect a SOCK_STREAM (tcp) socket to any other gnutella server, send a GNUTELLA CONNECT/0.4[[ff][ff] and expect back a GNUTELLA OK[[ff][ff]. At this point the server expects you to identify yourself. You send a type 0x00 message to whoever you just connected to, and the server responds with how many files it is sharing, and the total size of those files (in KB). You'll also get a response from everybody connected to the machine you connect to, and so on, until the TTL expires on the message.

At this point, the server will start bombarding you with information about other servers (which fills the host catcher, and gnutellanet stats). You'll also get search requests. You're supposed to decrement the TTL and pass it on to any other servers you're connected to (if TTL > 0) If you have no matching files you can simply discard the packet, otherwise you should build a query response to that message and send it back from where it came.

The header is fixed for all message and ends with the size of the data area which follows. The header contains a Microsoft GUID (Globally Unique Identifier for you nonWinblows people) which is the message identifier. My crystal ball reports that "the GUIDs only have to be unique on the client", which means that you can really put anything here, as long as you keep track of it (a client won't respond to you if it sees the same message id again). If you're responding to a message, be sure you haven't seen the message id (from that host) before, copy their message ID into your response and send it on its way. That message ID is followed by a function ID (one byte), which looks to be a bitmask. The function ID indicates what type to do with the packet (search request, search response, server info, etc). The next field is a byte TTL. Every packet you receive you should dec (or -- for the C guys) the TTL and pass the packet on if the TTL is still > 0 (i.e. if (--hdr.TTL) { [pass on] }, god I love C). You should also inc the hop count. Seems redundant? Well, some people have smaller TTLs, and you have the right to drop any message you want to based on its hop count. The header finishes up by telling us how large the function-dependant data that follows is.

Searches:

Easy, just build a type 0x80 packet, add a WORD for the minimum connection speed(in kbps), then the null terminated string. There isn't a response from people who have no match, but a result will come back as a type 0x81 message. There will be a Search Response header followed by N Search Response Items and double NULL terminated filenames. To finish this up, there's a Search Response footer with the full 128 bit (16 byte) client ID of the server that found the result.

Downloads and Uploads:

These are POC. If you want a file from a server, you connect to the server, and send an HTTP request for it. The URL is of the form /get[/file_id][/filename]. The file id was returned with the search result. The gnutella HTTP server also supports resuming a transfer via the Content-range: HTTP header. If you're just curious, the User-Agent is gnutella. You can actually load up Netscape, and get a file from a Gnutella server. Pretty cool, eh? Here's a dump of what a HTTP request looks like:

GET /get/293/rhubarb_pie.rcp HTTP/1.0
User-Agent: gnutella

Yes, the user-agent header and HTTP version are required. If the server is behind a firewall which does not allow incoming connections, the client can negotiate a push connection. This is a function ID 0x40 packet. It contains the ClientID128 (GUID) of the server, followed by the File ID requested, and the IP address and port of the client.

Awright, some people have read this document and sill keep asking me things like "What's in the first 10 bytes of the header?", so for people who can't figure out how what the syntax of a Delphi record looks like, or can't read english so well, here's some nice tables:

**Header:**

**Bytes 0 - 15:** Message ID:
A message ID is generated on the client for each new message it creates. The 16 byte value is created with the Windows API call CoCreateGuid(), which in theory will generate a new globally unique value every time you call it. See the text above for a comment about this values uniqueness.

**Byte 16:** Function ID:
What message type the packet is. See the table of message types below for descriptions of the types.

**Byte 17:** TTL Remaining:
How many hops the packet has left before it should be dropped.

**Byte 18:** Hops taken:
How many hops this packet has already taken. Set the TTL on response messages to this value!

**Bytes 19 - 22:** Data Length:
The length of the Function-dependant data which follows. There has been some discussion as to if this value is actually only 2 bytes and the last 2 bytes are something else. Seems to work with 4 for me. Also there is a question as to signed or unsigned integers. Don't know that either, I can't get gnutella to try and send a 2^31 + 1 byte packet :).

**Function IDs:**

0x00: **Ping:**
An empty message (datalen = 0) sent by a client requesting an 0x01 from everyone on the network. This message type should be responded to with a 0x01 and passed on.

0x01: **Ping Response:**
Sent in response to a 0x00, this message contains the host ip and port, how many files the host is sharing and their total size.

0x40: **Client Push Request:**
For servants behind a firewall, where the client cannot reach the server directly, a push request message is sent, asking the server to connect out to the client and perform an upload.

0x80: **Search:**

This is a search message and contains the query string as well as the minimum speed.

0x81: Search Response:
These are results of a 0x80 search request it contains the IP address, port, and speed of the serverant, followed by a list of file sizes and names, and the ClientID128 of the serverant which found the files. ClientID128 is another 16 byte GUID. However, this GUID was created once when the client as installed, is stored in the gnutella.ini, and never changes.

Ping:
A Ping has no body.

Ping Response:

Bytes 23 - 24: Host port:
The TCP port number of the listening host

Bytes 25 - 28: Host IP:
The IP address of the listening host, in network byte order.

Bytes 29 - 32: File Count:
An integer value indicating the number of files shared by the host. No idea if this is a signed or unsigned value.

Bytes 33 - 36: Files Total Size
An integer value indicating the total size of files shared by the host, in kilobytes (KB). No idea if this is a signed or unsigned value.

Search:

Bytes 23 - 24: Minimum speed:
The minimum speed of serverants which should perform the search. This is entered my the user in the "Minimum connection speed" edit box.

Bytes 25 +: Search query:
A NULL terminated character string which contains the search request

Search Response:

Byte 23: Num Recs:
Number of Search Response Items which follow this header.

Bytes 24 - 25: Host Port:
The listening port number of the host which found the results.

Bytes 26 - 29: Host IP:
The IP address of the host which found the results. In network byte order.

Bytes 30 - 33: Host Speed:
The speed of the host which found the results. **This may be incorrect. I would assume that only 2 bytes would be needed for this. The last 2 bytes may be used to indicate something else.**

Bytes 34 +: List of Items:
A Search Response Item for each result found.

Last 16 bytes: Footer:
The clientID128 of the host which found the results. This value is stored in the gnutella.ini and is a GUID created with CoCreateGUID() the first time gnutella is started.

**Search Response Item:**

**Bytes 0 - 3: File Index:**
Each file indexed on the server has an integer value associated with it. When gnutella scans the hard drive on the server a sequential number is given to each file as it is found. This is the file index.

**Bytes 4 - 7: File Size:**
The size of the file (in bytes).

**Bytes 8 +: File Name:**
The name of the file found. No path information is sent, just the file's name. The filename field is double-NULL terminated.

**Push Request:**

**Bytes 23 - 38: ClientID128:**
The ClientID128 GUID of the server the client wishes the push from.

**Bytes 39 - 42: File Index:**
Index of file requested. See query_response_rec for more info.

**Bytes 43 - 46: Requester IP:**
IP Address of the host requesting the push. Network byte order.

**Bytes 47 - 48: Requester Port:**
Port number of the host requesting the push.

**Routing:**

An issue everyone wants to ask me about nowadays is routing. "Do I forward every packet I see to every connected host?" Holy Jesus no! That would swamp the network with duplicate packets (which it already s). Here's the secret. **For simplicity sake, TTL is not discussed in this section**

(Forgive the non-straight lines, but the internet's like that)

Imagine yourself as node 1 in the above diagram. You have direct gnutellanet (physical socket) connections to nodes 2, 3, 4, and 5. You have reachable hosts at nodes 6 thru 13.

1. You get an ping message (function 0x00) from 2 with a message id of x.
2. Lookup in your message routing table [message x, socket ???]
4. Respond with an Ping Response (0x01), message id x to node 2.
5. Send the function 0x00 message to nodes 3, 4, and 5 (not 2!!).
6. Node 3 will respond with Ping Response (0x01), message id x.
7. Forward the message to whoever in the list has [message x, socket ???], since this packet is being routed and not broadcast, there is no need to check for if it is a duplicate, as routed messages don't make loops.
8. Do the same thing with responses from 4 and 5.
9. Since 3 thru 5 will also pass the message on to 8 thru 13, you'll also get a 0x01 from them too.
10. **Problem:** Node 3 is connected to 10 who is connected to 4 who is connected to you! It's OK! You lookup in your route list [message x, socket ???]... It's already there! You drop the message, do not respond to 4, do not forward to anyone!

Here's the basic mechanics, described in the example above:

- If the low bit of the function ID (f) is 0, look for [message x, socket ???]. If it's already there, drop the message. If it isn't, add it as [message x, socket s], respond to socket s, and forward to all connected clients except socket s (the one you got the message from).
- If the low bit of the function ID (f) is 1, look for the socket which matches [message x, socket ???] and forward the message to that connection **only**.
- If the low bit of the function (f) is not 0 or 1, you need to stop letting an infinite number of monkeys use your machine while they work on their Hamlet script.

**TTL and Hops:**

"How many computers the packet can go through before it will stop being passed around like a whore." - Nouser (#gnutella on efnet)

TTL, anyone who knows anything about TCP/IP will tell you that TTL stands for Time To Live. Basically, when a packet (or message in our case) is sent out, it is stamped with a TTL, for each host that receives the packet, they decrement the TTL. If the TTL is zero, the packet is dropped, otherwise it is routed to the next host in the route. Gnutella TTLs work similarly. When a NEW message is sent from your host, the TTL is set to whatever you have set in your Config | TTL | My TTL setting. When the packet is received by the next host in line the TTL is decremented. Then that TTL is checked against that host's Config | TTL | Max TTL setting. The lower of the two numbers in placed in the outgoing TTL field. If the outgoing TTL is zero, the packet is dropped. [Capn's Note: I'm not positive about this next part.] Then the Hops field of the message is incremented and checked. If this number is greater than the Max TTL setting, the packet is dropped. [End Capn's Note.] This method means that even if you set your TTL to 255 (maximum value), odds are the TTL will be set to the default (5) by the next host in your chain.

This document originally written by CapnBry, bmayland@leoninedev.SPAM.com, and was downloaded from http://capnbry.dyn dns.org/gnutella/protocol.php.

Home: Docs

SYSTEM, METHOD, AND COMPUTER PROGRAM FOR
FLOW CONTROL IN A DISTRIBUTED BROADCAST-ROUTE NETWORK WITH
RELIABLE TRANSPORT LINKS

Inventor:
Serguei Y. Osokine

Field of the Invention
This invention pertains generally to systems and methods for communicating information
over an interconnected network of information appliances, and more particularly to system and
method for controlling the flow of information over a distributed information network having
broadcast-route network and reliable transport link network characteristics.

BACKGROUND
The Gnutella network does not have a central server and consists of the number of equal-
rights hosts, each of which can act in both the client and the server capacity. These hosts are
called ‘servents’. Every servent is connected to at least one other servent, although the typical
number of connections (links) should be more than two (the default number is four). The resulting
network is highly redundant with many possible ways to go from one host to another. The
connections (links) are the reliable TCP connections.

When the servent wishes to find something on the network, it issues a request with a
globally unique 128-bit identifier (ID) on all its connections, asking the neighbors to send a
response if they have a requested piece of data (file) relevant to the request. Regardless of
whether the servent receiving the request has the file or not, it propagates (broadcasts) the request
on all other links it has, and remembers that any responses to the request with this ID should be
sent back on the link which the request has arrived from. After that if the request with the same
ID arrives on the other link, it is dropped and no action is taken by the receiving servent in order to avoid the 'request looping' which would cause an excessive network load.

Thus ideally the request is propagated throughout the whole Gnutella network (GNet), eventually reaching every servent then currently connected to the network. The forward propagation of the requests is called 'broadcasting', and the sending of the responses back is called 'routing'. Sometimes both broadcasting and routing are referred to as the 'routing' capacity of the servent, as opposed to its client (issuing the request and downloading the file) and server (answering the request and file-serving) functions. In a Gnutella network each node or workstation acts as a client and as a server.

Unfortunately the propagation of the request throughout the whole network might be difficult to achieve in practice. Every servent is also the client, so from time to time it issues its own requests. Thus if the propagation of the requests is unlimited, it is easy to see that as more and more servents join the GNet, at some point the total number of requests being routed through an average servent will overload the capacity of the servent physical link to the network.

Since the TCP link used by the Gnutella servents is reliable, this condition manifests itself by the connection refusal to accept more data, by the increased latency (data transfer delay) on the connection, or by both of these at once. At that point the Gnutella servent can do one of three things: (i) it can drop the connection, (ii) it can drop the data (request or response), or (iii) it can try to buffer the data in hope that it will be able to send it later.

The precise action to undertake is not specified, so the different implementations choose different ways to deal with that condition, but it does not matter – all three methods result in serious problems for the Gnet, namely one of A, B, or C, as follows: (A) Dropping the connection causes the links to go up and down all the time, so many requests and responses are simply lost, because by the time the servent has to route the response back, the connection to route it to is no longer available. (B) Dropping the data (request or response) can lead to a response being dropped, which overloads the network by unnecessarily broadcasting the requests over hundreds of servents only to drop the responses later. (C) Buffering the data increases the latency even more. And since it does little or nothing to fix the basic underlying problem (an attempt to transmit more data than the network is physically capable of) it only causes the servents to
eventually run out of memory. To avoid that, they have to resort to other two ways of dealing with the connection overload albeit with much higher link latency.

These problems were at least somewhat anticipated by the creators of the Gnutella protocol, so the protocol has a built-in means to limit the request propagation through the network, called ‘hop count’ and ‘TTL’ (time to live). Every request starts its lifecycle with a hop count of zero and TTL of some finite value (de facto default is 7). As the servent broadcasts the request, it increases its hop count by one. When the request hop count reaches the TTL value, the request is not broadcast anymore. So the number of hosts N that see the request can be approximately defined by the equation:

\[
N = (avLinks - 1)^{\text{TTL}}, \quad \text{(EQ. 1)}
\]

where \( avLinks \) is the average number of the servent connections, and the TTL is the TTL value of the request. For the \( avLinks = 5 \) and \( TTL = 7 \) this comes to a value of \( N \) of about 10,000 servents.

Unfortunately the TTL value and the number of links are typically hard-coded into the servent software and/or set by the user. In any case, there’s no way for the servent to quickly (or dynamically) react to the changes in the GNet data flow intensity or the data link capacity. This leads to the state of affairs when the GNet is capable of functioning normally only when the number of servents in the network is relatively small or they are not actively looking for data. When either of these conditions is not fulfilled, the typical servent connections are overloaded with the negative consequences outlined elsewhere in this description. Put simply, the GNet enters the ‘meltdown’ state with the number of ‘visible’ (searchable from the average servent) hosts dropping from the range of between about 1,000-4,000 to a much smaller range or between about 100-400 or less, which decreases the amount of searchable data by a factor of ten or about an order of magnitude. At the same time the search delay (the time needed for the request to traverse 7 hops (the default) or so and to return back as a response) climbs to hundreds of seconds. Response time on the order of hundreds of seconds are typically not tolerated by users, or at the very least are found to be highly irritating and objectionable.

In fact, the delay becomes so high that the servent routing tables (the data structures used to determine which connection the response should be routed to) reach the full capacity, overflow
and time out even before the response arrives so that no response is ever received by the requestor. This, in turn, narrows the search scope even more, effectively making the Gnutella unusable from the user standpoint, because it cannot fulfill its stated goal of being the file searching tool.

5

The 'meltdown' described above has been observed on the Gnutella network, but in fact the basic underlying problem is deeper and manifests itself even with a relatively small number of hosts, when the GNet is not yet in an actual meltdown state.

10 The problem is that the GNet uses the reliable TCP protocol or connection as a transport mechanism to exchange messages (requests and responses) between the servants. Being the reliable vehicle, the TCP protocol tries to reliably deliver the data without paying much attention to the delivery latency (link delay). Its main concern is the reliability, so as soon as the data stream exceeds the physical link capacity, the TCP tries to buffer the data itself in a fashion, which is not controlled by the developer or the user. Essentially, the TCP code hopes that this data burst is just a temporary condition and that it will be able to send the buffered data later.

15 When the GNet is not in a meltdown state, this might even be true – the burst might be a short one. But regardless of the nature of the burst, this buffering increases the delay. For example, when a servant has a 40 kbit/s modem physical link shared between four connections, every connection is roughly capable of transmitting and receiving about 1 kilobyte of data per second. When the servant tries to transmit more, the TCP won't tell the servant application that it has a problem until it runs out of TCP buffers, which are typically of about 8 kilobyte size.

20

So even before the servant realizes that its TCP connections are overloaded and has any chance to remedy the situation, the link delay reaches 8 seconds. Even if just two servants along the 7-hop request/response path are in this state, the search delay exceeds 30 seconds (two 8-second delays in the request path and two – in the response path). Given the fact that the GNet typically consists of the servants with very different communication capabilities, the probability is high that at least some of the servants in the request path will be overloaded. Actually this is exactly what can be observed on the Gnutella network even when it is not in the meltdown state

25

86
despite the fact that most of the servents are perfectly capable of routing data with a sub-second delay and the total search time should not exceed 10 seconds.

Basically, the 'meltdown' is just a manifestation of this basic problem as more and more servents become overloaded and eventually the number of the overloaded servents reaches the 'critical mass', effectively making the GNet unusable from a practical standpoint.

It is important to realize that there's nothing a servent can do to fight this delay – it does not even know that the delay exists as long as the TCP internal buffers are not yet filled to capacity.

Some developers have suggested that UDP be used as the transport protocol to deal with this situation, however, the proposed attempts to use UDP as a transport protocol instead of TCP are likely to fail. The reason for this likely failure is that typically the link-level protocol has its own buffers. For example, in case of the modem link it might be a PPP buffer in the modem software. This buffer can hold as much as 4 seconds of data, and though it is less than the TCP one (it is shared between all connections sharing the physical link), it still can result in a 56-second delay over seven request and seven response hops. And this number is still much higher than the technically possible value of less than ten seconds and, what is more important, higher than the perceived delay of the competing Web search engines (such as for example AltaVista, Google, and the like), so it exceeds the user expectations set by the 'normal' search methods.

Therefore, there remains a need for a system, method, and computer program and communication protocol that minimizes the latency and reduces or prevents GNet or other distributed network overload as the number of servents grows.

**SUMMARY**

The invention provides improved data or other information flow control over a distributed computing or information storage/retrieval network. The flow, movement, or migration of information is controlled to minimize the data transfer latency and to prevent overloads. A first or outgoing flow control block and procedure controls the outgoing flow of data (both requests and responses) on the network connection and makes sure that no data is sent before the previous portions of data are received by a network peer in order to minimize the
connection latency. A second or Q-algorithm block and procedure controls the stream of the requests arriving on the connection and decides which of them should be broadcast to the neighbors. Its goal is to make sure that the responses to these requests would not overload the outgoing bandwidth of this connection. A third or fairness block makes sure that the connection is not monopolized by any of the logical request/response streams from the other connections. It allows to multiplex the logical streams on the connection, making sure that every stream has its own fair share of the connection bandwidth regardless of how much data are the other streams capable of sending. These blocks and the functionality they provide may be used separately or in conjunction with each other. As the inventive method, procedures, and algorithms may advantageously be implemented as computer programs, such as computer programs in the form of software, firmware, or the like, the invention also advantageously provides a computer program and computer program product when stored on tangible media. Such computer programs may be executed on appropriate computer or information appliances as are known in the art, and may typically include a processor and memory couple to the processor.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a diagrammatic illustration showing an embodiment of a distributed information network providing flow control according to the invention.

DETAILED DESCRIPTION OF EMBODIMENTS OF THE INVENTION

The inventive system, method, and computer program solve the aforesaid described problems and limitations by minimizing the latency and preventing Gnutella and non-Gnutella distributed network overload as the number of nodes (clients, servers, or servants) grows. Almost all features of the inventive method and algorithm, except possible for some Gnutella specific version backward compatibility features, are not Gnutella-specific and can be utilized by any distributed network of similar architecture or topology. An exemplary embodiment of the invention is illustrated in FIG. 1 which shows an embodiment of a distributed information network on the form of a Gnutella network providing flow control according to the invention.

Aspects of the inventive system and method are directed to achieving large and in practical terms nearly infinite scalability of the distributed networks, which use a broadcast-route or analogous method to propagate the requests (such as information, data, or file requests) through the network. The broadcast-route as used here means a method of request propagation
when the host broadcasts the request it receives on every connection it has except the one it came from and later routes the responses back to that connection.

The inventive method and algorithm is designed for the networks with reliable transport links. This primarily means the networks, which use TCP and TCP-based protocols (i.e. HTTP) for the data exchange, but the usefulness of the algorithm is not limited to TCP-based networks, and those workers having ordinary skill in the art will appreciate the applicability of the inventive system, method, and algorithm to other communication and signaling protocols that are expected to be developed and adopted in the foreseeable future. The method and algorithm might be used even for the ‘unreliable’ transport protocols like UDP, because they might (and do) utilize the reliable protocols on some level – for example, unreliable UDP packets can be sent over the modem link with the help of the reliable (PPP or its analog) protocol. Finally, as already indicated, the primary target of the algorithm is the Gnutella network (GNet), which is widely used as the distributed file search and exchange system.

The inventive system and method achieve large and in practical terms nearly infinite scalability of the distributed computing and information exchange networks, which use the ‘broadcast-route’ method to propagate the requests through the network. The ‘broadcast-route’ here means the method of the request propagation when the host broadcasts the request it receives on every connection it has except the one it came from and later routes the responses back to that connection. The inventive algorithm is designed for the networks with reliable transport links. This primarily means the networks, which use TCP and TCP-based protocols (i.e. HTTP) for the data exchange, but the usefulness of the algorithm is not limited to TCP-based networks. The algorithm might be used even for the ‘unreliable’ transport protocols like UDP, because they might (and do) utilize the reliable protocols on some level – for example, unreliable UDP packets can be sent over the modem link with the help of the reliable (PPP or its analog) protocol.

In spite of the generality of its applicability, the primary target of the algorithm is the Gnutella network, which is widely used as the distributed file search and exchange system. Its protocol specifications can be found as of 28 November 2000 on the Internet at:
http://gnutella.wego.com/go/wego.pages.page?groupId=116705&view=page&pageId=119598&folderId=116767&panelId=-1&action=view;
http://www.gnutelladev.com/docs/capnbry-protocol.html;
The system, method, and algorithm dealing with the problems described above should satisfy several conditions: (i) It should make the Get highly scalable and in a practical sense as infinitely scalable as possible — that is, the large or practically infinite growth of the number of hosts in the network should not result in the network meltdown. (ii) It should bring the search delay to a technically reasonable minimum — that is, it should not result in an excessive request/response buffering when it is avoidable. (iii) It must be able to be used to connect the servants with various physical link capacities (even including asymmetrical links with different upload and download bandwidths where feasible) without causing the overload of the low-capacity links by the high-capacity ones. (iv) It should make a best-effort attempt to fairly share the connection between the logical request/response streams from other connections with different bandwidths, preventing the ‘starvation’ of the low-bandwidth streams because of the connection being monopolized by the high-rate logical streams. That also includes the ability of the algorithm to survive the denial-of-service (DoS) attacks, which are essentially the attempts of a single host or a group of hosts to monopolize the network, denying service for the legitimate users. (v) It should be backward-compatible with the already deployed Gnutella servant code base — even though the ‘old’ servants might be unable to fully reap the benefits of the algorithm, at the very least, they should be able to work with the ‘new’ servants without any quality of service degradation.

Actually almost all of the inventive method and procedure’s features (except for the backward compatibility) are not Gnutella-specific and can be utilized by any distributed network with a similar ‘broadcast-route’ architecture as long as the network transport mechanisms include the reliability feature at least on one level.

One algorithmic or methodological goal of the invention is to control the flow of data through the Gnutella servant in order to minimize the data transfer latency and to prevent overloads, so it is called the ‘flow control’ algorithm. It consists of three basic blocks. A first or "outgoing flow control block" controls the outgoing flow of data (both requests and responses) on the connection and makes sure that no data is sent before the previous portions of data are received by the peer servant in order to minimize the connection latency. A second or "Q-
algorithm block" controls the stream of the requests arriving on the connection and decides which
of them should be broadcast to the neighbors. Its goal is to make sure that the responses to these
requests would not overload the outgoing bandwidth of this connection. A third or "fairness
block" makes sure that the connection is not monopolized by any of the logical request/response
streams from the other connections. It allows to multiplex the logical streams on the connection,
making sure that every stream has its own fair share of the connection bandwidth regardless of
how much data are the other streams capable of sending. These blocks and the functionality they
provide may be used separately or in conjunction with each other. For example, the flow control
block may be used without the Q-algorithm block or the fairness block, and both the flow control
block and the q-algorithm block may be used together without the fairness block. Each of the
structural and algorithm features of these blocks are described in greater detail below.

Outgoing Flow Control Block

The Outgoing Flow Control Block controls the flow of data sent to a connection which
has reliable transport at least somewhere in its network stack and tries to minimize the latency
(transport delay) of this connection. It recognizes that some components of the transport delay
cannot be controlled (physical limit determined by the speed of light, transport delay on the
Internet routers, and the like). Still, the delays related to the buffering undertaken by the reliable
transport layers (TCP, PPP, or the like) can be several times higher (seconds as opposed to
hundreds of milliseconds), so their minimization can have dramatic effects on the transport delay
on the connection.

The flow control block tries to use the simplest possible way of controlling the delay in
order to decouple itself as much as possible from the specific buffering and retransmission
mechanisms in the reliable transport, even at the cost of some theoretical connection throughput
loss.

Thus several goals are achieved by the outgoing flow control block as now described.
First, the algorithm can be used over a variety of transport mechanisms – not only TCP, but also
HTTP over TCP, UDP over PPP, etc. This makes it possible to use the same algorithm if for
some reason (for example, firewalls or something else) it would be necessary to migrate the
distributed network "broadcast-route" algorithm to a different transport protocol. Second, it
makes the flow control algorithm not sensitive to a specific TCP implementation. Even though it
is possible to optimize the flow control algorithm for an 'ideal' TCP implementation, it might be
difficult to verify such an algorithm for every existing implementation of TCP, requiring the
complicated and unreliable compatibility testing. Thus independence from a specific TCP
implementation is valuable. Third, the 'decoupling' of the flow control algorithm from the
underlying transport layer minimizes the possible feedback effects between it and the transport,
making the algorithm more stable from a control theory standpoint.

In fact, very few assumptions are made about the nature of the underlying transport layers
– even the reliability is not required. The algorithm mitigates possible negative effects of the
reliable transport layer (that is, minimizes buffering) without relying on the help it might get from
this layer. Thus we can migrate the flow control algorithm to UDP or use it to communicate with
the clients who know nothing of this algorithm or the ones with an erroneous code, so not every
request and response is delivered between them.

In order to achieve all these goals, the algorithm uses the simplest 'zig-zag' data receipt
confirmation between the clients on the application level of the protocol. In case of the Gnutella
network, PING messages with a TTL (time to live) of 1 are inserted into the outgoing data stream
at the regular intervals (every 512 bytes). Such messages are broadcast only over 1-hop distance,
so the peer should not broadcast them – it should just reply to them with PONGs (PING response)
messages.

As soon as the sender receives such a response, the sender can be sure that all the data
preceding the PING message has reached its intended destination, so it can send another 512
bytes with a PING. Thus we can be sure that at no moment the buffers of all the networking
layers between servants contain more than 512 bytes+PING size + PONG size. Since the PING
size in the Gnutella protocol is 23 bytes and the PONG size is 37 bytes, we are wasting only
about 10.5% (See Eq. 2) of the whole bandwidth on these transmissions, which is not a huge
amount, considering that the zig-zag schema occupies only about ½ of the available bandwidth in
any case.

\[
(2) \quad \frac{\text{PINGsize}+\text{PONGsize}}{(\text{512}+\text{PINGsize}+\text{PONGsize})} = \frac{60}{572} = 10.5 \% \quad (\text{EQ. 2})
\]
This 'under-utilization' of the bandwidth is actually not an undesirable feature of the inventive method and algorithm -- would it happen to occupy the whole bandwidth, other applications (Web browsing, file upload/download) on the same computer would suffer. The flow control algorithm recognizes the fact that the distributed network routing servent is designed to work in the background with as little interference with the more important computer tasks as possible, so it tries to be unobtrusive.

The inventive method and algorithm also recognizes that the peer servent might know nothing of the flow control and/or be buggy (that is contain errors or bugs), so it might lose the PING altogether, might forward it to peers, sending back multiple PONG responses and so on. So the unique global ID field of the PING message is used to 'mark' the outgoing PINGS with their position in the connection data stream. Thus the duplicate and out-of-order PONGs can be rejected. Furthermore, if the reply does not arrive in 1 second (or some other predetermined period of time), the new PING (with a new sequence number) is sent in case the previous request or response was lost.

The inventive method and algorithm also limits the connection latency. For example, a client with a 33-40 kbits/sec modem and five connections will never have more than 3 Kbytes of data in transmission (572 times 5 plus headers), so even for this slow client the request delay won't be much more than about 900 ms (See EQ. 3) which is several times less than the typical latency value for the GNet without the flow control.

\[
(3) \quad 3 \text{Kbytes} \times 10 \text{bits/byte} / 33,000 \text{kbits/sec} = 900 \text{ms} \quad \text{(EQ. 3)}
\]

At the same time the inventive method and algorithm effectively limits the maximum possible data transfer rate on the connection. For example, if the Internet network round-trip time between servents is 150 ms, the connection won't be able to send more than 3400 bytes of data per second (140 kbits/sec per physical link if the servent has 5 connections) regardless of the physical link capacity. This feature can be viewed as a desirable one, but if it is not, this can be remedied by other means, such as for example by opening the additional connections, or by increasing the 512-byte data threshold between PINGs.
Note that if the underlying transport is TCP, the developer may be well advised to switch off the Nagle algorithm and send all the data in one operation (as one packet). Even though the algorithm will function in any case, this might allow the servant to get rid of the 200-ms additional latency added by the Nagle algorithm to the TCP stream. Since this might lead to an excessive packet traffic in case of the short roundtrip times and low sending rates, the outgoing flow control block implements the ‘G-Nagle’ algorithm. This is essentially a 200-ms timer, which is started when the packet is sent out. It prevents the next packet sending operation if all three of the following conditions are true: (i) the packet is not fully filled (less than 512 bytes), (ii) the previous packet RTT echo has already returned, and (iii) the 200-ms timeout has not expired yet. This algorithm is called ‘G-Nagle’ since it is basically a Nagle TCP algorithm ported into the Gnutella context. It serves the same goal of limiting the traffic with low payload-to-headers ratio if it is possible to do so without causing the perceptible latency increase.

Naturally, the flow control block routinely has to deal with the situations when other connections want to transmit some data, which cannot be sent because the PONG echo has not arrived yet and the 512-byte packet is already filled. In this case the behavior of the algorithm is determined by whether this data is the request (a broadcast) or the response (a routed-back message). The responses, which do not fit into the bandwidth, are buffered for the future sending — the goal is to keep the link latency to a minimum. It is the job of the Q-algorithm (described below) to make sure that these responses will be eventually sent out. The requests are just dropped according to their current hop count — the less is the hop count, the more important the request is considered to be.

This prioritization of the requests has two goals — first, it tries to make sure that the ‘new’, low-hop requests, which still exist only in a few copies on the network, are not dropped. If the zero-hop request is dropped, we lose a significant percent of this request copies and statistically are quite likely to drop all of them, thus making the whole search unsuccessful. And second, by dropping the high-hop requests, it tries to effectively introduce something like ‘local TTL’ into the network. We are effectively dropping the request with the hop count higher than some limit, but, unlike TTL, this limit is not preset or hard-coded. It is determined at run-time by the congestion state of the neighboring network segment. When we are doing this, we want to keep the average length of the search route to a minimum in order to minimize the network load and latency arising from the packet duplication on the way back. For example, it places a lighter
load on the network to reach 4 servents with a 1-hop request over 4 connections, than to reach the
same number of hosts with a 4-hop request over one connection. (The broadcast load is the same,
but the first case results in 4 response messages with, say, 1-second latency, and the second case
results in $1+2+3+4=10$ messages transmitted over the network with a total latency of 4 seconds.)

Actually the precise algorithm defining which requests should be sent out and which
should be dropped may be made a bit more complicated due to the competition between the
requests coming from different connections. This scenario is further developed in the description
of the Fairness block elsewhere in this description. The outgoing flow control block main goal is
the latency minimization of the connection – not the fair connection bandwidth sharing between
different logical substreams. However, it is worth mentioning that the TTL value of the Gnutella
protocol loses its flow-control purpose in the presence of the flow control algorithm. It is
desirably used mainly to avoid an infinite packet looping in case the servent code contains errors
– pretty much in the same fashion as the IP packet TTL field is used by the Internet protocol (IP)
networking layer code.

The responses (back-traffic) are also prioritized, but according to the total number of
responses routed for their globally unique request ID. The idea is to pass the small number of
responses at the highest priority without any delay, but if the request was formulated in a way,
which resulted in a large number of responses, it is only fair to make the requesting servent wait.
Since this request places an extra load on the network above the average level, it makes sense to
send back its responses at the lowest priority only after the ‘normal request’ responses are sent.

The Q-Algorithm Block

The Q-algorithm block is logically situated on the receiving end of the connection. Its job
is to analyze the incoming requests and to determine which of them should be broadcast over
other connections (neighbors), and which should be dropped and not even passed to the outgoing
flow control algorithm block described above.

The Q-algorithm block tries to limit the flow of broadcast requests so that the responses
would not overflow the outgoing bandwidth of the connection, which has supplied these requests
and will have to accept responses. Every response to route is typically a result of some significant
effort of the network to broadcast the request, to route back the responses, and it would be wrong
to drop the responses in the outgoing flow control block – it is more prudent just not to broadcast
the requests. Besides, every response is a unique answer of some servant to some request. If the
request is dropped, the redundant distributed network still might deliver the request to that servant
over some other route with sufficient bandwidth. But if the response is dropped, the answer is
irretrievably lost – the servant won’t respond again, since it has already seen the request with this
Global Unique ID (GUID).

So it is a job of the Q-algorithm block and the method provided therein to control the
flow of the incoming requests so that the responses would not overflow the outgoing bandwidth
of the same connection. It is especially important when the typical request generates several
responses (highly available file is found) and/or the physical link is asymmetric – the connection
cannot send as much data out as it can receive, which is typical for the ADSL and other high-
bandwidth consumer-oriented ISP solutions. The excessive requests (the ones, which, if sent,
would overflow the outgoing connections) are dropped according to essentially the same hop-
priority algorithm as the one used by the outgoing flow control block.

Here it is important to note that in no case should Q-algorithm try to control the flow of
the incoming requests x by slowing down the network reading. In fact, such an approach would
likely be fatal for the flow control algorithm as a whole. First, it would ruin the algorithm
backward compatibility in case of the Gnutella network – the flow control-unaware servants
would keep sending data to the connection, which would be read with a slower rate. This would
result in an explosive latency growth on the connection and possibly the connection would be
dropped by the sender altogether.

Second, even in case of the flow control-compliant servants, who would use their
outgoing flow control blocks to limit the connection latency, it should be remembered that the
connection is shared between several streams of a very different nature. For example, the same
connection is used to transfer both requests (which can be just dropped if the bandwidth is tight)
and responses (which have to be delivered in any case to avoid the useless loading of the
network). When the connection reading rate is throttled down, not only does the rate of the
broadcasts from that connection decrease, but the rate of the response routing in the same
direction also decreases. This response rate is something, which is supposed to be determined by
the Q-algorithm on another servant and represents this other servant connection’s back-stream. So
there's no good reason why we should punish this back-stream just because our side of this connection has to throttle down the back-stream in the opposite direction.

Because of this, the output forward-stream flow $x$ of the Q-algorithm should be formed as the subset of the incoming forward-stream $f$ (requests, arriving to the connection). These requests should be read as fast as possible and only then those of them which exceed the required $x$ value should be dropped. Unfortunately it may be very difficult to directly control the back-stream of responses by controlling the forward-stream of the requests. The direct application of the control theory fails for a variety of reasons, the main one being that the response for any particular request is not known in advance and its size and delay cannot be known with any degree of accuracy. The same request can result in zero responses and one hundred responses. The answers can arrive at once or be distributed over a 100-second interval – the rate cannot be predicted with any certainty. Besides, the volume of the back-traffic (responses) does not depend linearly on the amount of the forward-traffic (requests). After a certain broadcast rate is achieved, the outgoing flow control blocks on the same and/or other servers might take over and the further increases in the forward traffic won't affect the back-traffic at all. All this makes the 'normal' control theory inapplicable in that case.

This is why, in practical terms, the best one can hope for is to achieve the statistical control over the response traffic – say, 'x' bytes per second of requests on the average generate 'b' bytes per second of back-traffic responses. The ideal case would be that the back-traffic can be averaged and as the averaging interval grows the relative error (variation of the back-traffic distribution) converges to zero. Unfortunately, for all practical purposes this zero (or close to zero) relative variation of the back-traffic cannot be achieved since it turns out that statistically the distributions of responses on the Gnutella network exhibit the clearly pronounced fractal properties.

The fractal character of many network- and traffic-related processes is a well-known fact, demonstrated by many researchers See, for example, "On the Self-Similar Nature of Ethernet Traffic (Extended Version)" by W. Leland, M. Taqqu, W. Willinger and D. Wilson in IEEE/ACM TRANSACTIONS ON NETWORKING, Vol. 2. No. 1. February 1994; which is hereby incorporated by reference. It has been shown that the fractal, or self-similar distributions can be the result of the superposition of many independently functioning random processes with a
'heavy-tailed' distributions – that is, the distributions which asymptotically follow a power law. That is,

\[ P[X > x] \sim x^{-\alpha} \text{ as } x \to \infty, \text{ where } 0 < \alpha < 2. \]  \tag{4}

(See, for example, "On the Effect of Traffic Self-similarity on Network Performance" by K. Park, G. Kim and M. Crovella in Proceedings of the 1997 SPIE International Conference on Performance and Control of Network Systems; which is hereby incorporated by reference).

Since the responses of Gnutella servants to the request are independent and the size of the response can range from zero (no matches are found) to a very large value (every servant responds), it is not a surprise that the back-traffic exhibits a fractal behavior.

From the mathematical standpoint, the self-similar random processes are characterized by the Hearst parameter \( H > \frac{1}{2} \) or the autocorrelation function of the form,

\[ r(k) \sim k^{-\beta} \text{ as } k \to \infty, \text{ where } 0 < \beta < 1. H = 1 - \beta/2. \]  \tag{5}

From the practical standpoint, the fractal traffic is characterized by its inherently bursty nature, which means that the bursts are clearly visible on very different time-scale plots and are not easily averaged as the averaging interval is increased (see the references above). It means that the back-traffic cannot be easily controlled so that its average would be close to the link (connection) bandwidth with a small variance. There's always a non-negligible chance that the traffic burst overloads the connection regardless of the averaging interval when a request with many responses is broadcast over the network. The results of the queuing theory suggesting that the mean latency starts to grow to infinity only as the link utilization (percent of the available bandwidth used to route back responses) approaches 1, do not apply for a fractal traffic. The latency growth starts with the loads much lower than the ones predicted by the queuing theory.

The results of the fractal traffic research (See for example, "Experimental Queuing Analysis with Long-Range Dependent Packet Traffic" by A. Erramilli, O. Narayan and W. Willinger; which is hereby incorporated by reference) suggest that an only way to keep the link
 latency acceptable is to keep the link utilization at about \( \frac{1}{2} \) level. That is, on the average, we should use only half of the link bandwidth to route the responses back.

It is to control this ‘average back-traffic’ that the Q-algorithm was introduced. Given the link bandwidth available for the back-traffic (responses) \( B \), it tries to bring the average back-traffic rate \( b \) to the value, which would occupy the predefined portion of the available bandwidth \( \rho \):

\[
(6) \quad <b> \rightarrow <\rho B>
\quad \text{(EQ. 6)}
\]

For the sufficiently small values of \( \rho \) (say, \( \rho \leq \frac{1}{2} \)) that assures that the majority of the responses will be able to be routed back to the request originator without any delay whatsoever. Only the traffic bursts exceeding the target back-traffic value of \( \rho B \) by a factor of \( 1/\rho \) (which is \( \geq 2 \)), will cause the connection overload and the response latency increase, so this increase will affect only the small percentage of the responses. And even then, because of the back-traffic prioritization performed by the outgoing flow control block, the latency will be increased mainly for the requests with an unusually high number of responses.

Since the servant cannot control the back-traffic directly - only by controlling the amount of the forward-traffic (requests) \( x \) that it broadcasts - it is convenient to write our goal (EQ. 6) as:

\[
(7) \quad <x R_{av}> \rightarrow <\rho B>, \quad \text{(EQ. 7)}
\]

where \( R_{av} \) is the estimated back-to-forward ratio; on the average, every byte of the requests passed through the Q-algorithm to be broadcast eventually results in \( R_{av} \) bytes of the back-traffic on that connection.

Furthermore, since the amount of the forward-traffic being broadcast is naturally limited by the rate of the requests arriving to the connection \( f \), \( x \leq f \). So it can be that even when every arriving request is broadcast, still \( x R_{av} = f R_{av} < \rho B \). This is why the new variable \( Q \) is introduced:

\[
(8) \quad Q = x R_{av} + u, \quad Q \leq <B> = B_{av}, \quad \text{where} \quad u = \max(0, Q - f R_{av}) \quad \text{(EQ. 8)}
\]
Here \( u \) is the estimated underload factor. When \( u > 0 \), even if the Q-algorithm passes all the incoming forward traffic to be broadcast, it is expected that the desired part of the back-traffic bandwidth (\( \rho^*B \)) won't be filled. \( u \) is introduced into the equation to limit the infinite growth of the variable \( x \) when even the full forward traffic \( f \) won't fill the desired back-bandwidth. It allows to achieve the convergence of \( x^*Rav \) to its desired value \( \rho^*B \) when \( f^*Rav \leq \rho^*B \) and limits its numerical growth with \( f^*Rav \) otherwise. The algorithm tries to make \( Q \) converge to \( \rho^*B + u \) if it is possible:

\[
(9) \quad <Q> \rightarrow <\rho^*B + u>, \quad Q \leq <B> \quad \text{(EQ. 9)}
\]

In order to achieve that, the Q-algorithm uses the stochastic differential equation:

\[
(10) \quad \frac{dQ}{dt} = - (\text{beta/tauAv})^*(Q - \rho^*B - u), \quad \text{where} \quad \text{(EQ. 10)}
\]

\[
(11) \quad Q = x^*Rav + u, \quad Q \leq Bav \quad \text{(EQ. 11)}
\]

\[
(12) \quad u = \text{max}(0, Q - f^*Rav) \quad \text{(EQ. 12)}
\]

to control the back-traffic. This equation causes the value of the variable \( Q \) to exponentially converge to the mean value of \( \rho^*B + u \) with a characteristic time of \( -\text{tauAv} \) when the feedback coefficient \( \text{beta} = 1 \).

Note that the equations (EQ. 10 - EQ. 12) contain the random variables which cannot be controlled by the algorithm, like \( B \), \( Rav \) and \( f \). The Q-algorithm was specifically designed to avoid having the random and widely varying variables (like \( u \)) in the denominator of any equation. Otherwise, the stochastic differential equations of the Q-algorithm would exhibit some undesirable properties, preventing the algorithm output \( Q \) from converging to its target value.

The equations (EQ. 10 - EQ. 12) use the following conventions:

- \( B \) – the link bandwidth available for the back-traffic.

- \( \rho \) – the part of the bandwidth occupied by the average back-traffic (1/2).

- \( \text{beta} = 1 \) – the negative feedback coefficient.
\( \tau_{\text{Av}} \) – the algorithm convergence time. The relatively large interval chosen from the practical standpoint (100 seconds or so - more on that later).

\( Q \) – the Q-factor, which is the measure of the projected back-traffic. It is essentially the prediction of the back-traffic. The algorithm is called the ‘Q-algorithm’ because it controls the Q-factor for the connection. \( Q \) is limited with \( <B> \) to avoid the infinite growth of \( Q \) when \( f^*Rav \leq \rho * B \) and to avoid the back-stream bandwidth overflow (to maintain \( x^*Rav \leq B \) in case of the forward-traffic bursts.

\( x \) – the rate of the incoming forward-traffic (requests) passed by the Q-algorithm to be broadcast on other connections.

\( f \) – the actual incoming rate of the forward traffic.

\( Rav \) – the estimated back-to-forward ratio; on the average, every byte of the requests passed through the Q-algorithm to be broadcast eventually results in \( Rav \) bytes of the back-traffic on that connection. This estimate is an exponentially averaged (with the same characteristic time \( \tau_{\text{Av}} \)) ratio of actual requests and responses observed on that connection (see (EQ. 14) below).

\( Bav \) – the exponentially averaged value of the back-traffic link bandwidth \( B \). (\( Bav = <B> \)).

\( u \) – the estimated underload factor. When \( u > 0 \), even if the Q-algorithm passes all the incoming forward traffic to be broadcast, it is expected that the desired part of the back-traffic bandwidth (\( \rho * B \)) won’t be filled. It is introduced into the equation to limit the infinite growth of the variable \( x \) and ensure that \( x \leq f \) in that case.

At any given moment the amount of incoming request traffic to pass to other connections is determined by the equation:

\[
(13) \quad x = \frac{(Q - u)}{Rav} = \frac{\min(f^*Rav, Q)}{Rav} \quad \text{(derived from (EQ. 10 - EQ. 12)).}
\]

(EQ. 13)
Note that $x$ from (EQ. 13) always obeys the rule $x \leq f$ – only the traffic actually arriving to the connection can be broadcast.

The predicted average back-to-forward ratio $R_{av}$ is defined as an exponentially averaged instant value of the back-to-forward ratio $R$ observed on the connection:

\begin{equation}
\frac{dR_{av}}{dt} = -\left(\frac{\beta}{\tau_{av}}\right) (R_{av} - R) \quad \text{(EQ. 14)}
\end{equation}

One goal of the Q-algorithm block as defined by the equations (EQ. 10 - EQ. 14) is not to provide the quick reaction time to the changes in the back-traffic. In fact, in many cases it would be counterproductive, since it would cause the quick oscillations of the passed-through forward traffic $x$, making the system less stable and predictable. On the contrary, the Q-algorithm tries to gradually converge the actual back-traffic to its desired value $\rho * B$, doing this slowly enough so that the Q-algorithm would be effectively decoupled from other algorithm blocks. This is done to increase the algorithm stability – for example, so that the non-linear factors mentioned above would not have a significant effect and would not cause the self-sustained traffic oscillations.

This determines the choice of the Q-algorithm averaging time $\tau_{av}$. First of all, this time should not be less than the actual time $\tau_{rtt}$ passing between the request being broadcast and the responses being received, since it would not make any sense from the control theory standpoint. The control theory suggests that when the controlled plant has the internal delay $\tau_{rtt}$, it is useless to try to control it with the algorithm with characteristic time less than that. Any controlling action will not have effect for at least $\tau_{rtt}$ and trying to achieve the control effect sooner will only lead to the plant instability.

However, the network reaction time $\tau_{rtt}$ can be small enough (seconds) and we might want to make $\tau_{av}$ much bigger than that for the reasons outlined above – from the purely fractal traffic standpoint, it is ideal to have an infinite averaging time. On the other hand, it is impractical to have an infinite or even very large averaging time, since it limits the speed of the algorithm reaction to the permanent changes in the networking environment. For example, when one of the connections is closed and reopened to another host with a different bandwidth, the random process $R$, which defines the instant back-to-forward ratio is irretrievably replaced by the new one with another theoretical mean value $<R>$. At this point we need the equations (EQ. 10 -
EQ. 14) to converge to the new state, which is possible only if the algorithm averaging time is much less than the average connection lifetime.

For the Gnutella network the connection lifetime is measured in hundreds of seconds, so the averaging time is chosen to be:

\[
\text{tauAv} = \max(\text{tauRtt}, \text{tauMax}), \quad \text{where} \quad \text{tauMax} = 100 \text{ sec.} \quad \text{(EQ. 15)}
\]

At the same time it is important to remember that in case the back-traffic overloads the connection and it starts to buffer the responses, the Q-algorithm can cut the flow of the back-traffic and to decrease the overload only after the \text{tauAv} time interval. In the meantime, if the overload is caused not by a short burst, but rather by a permanent network state change, the back-traffic will continue to be buffered, increasing the effective response delay. In order to mitigate this, the averaging interval \text{tauAv} is made as small as reasonably possible (that is, \text{tauRtt}) when the mean back-traffic \(\langle b \rangle\) exceeds the mean connection bandwidth \(\langle B \rangle\).

The mean values for the back-traffic and for the connection bandwidth are approximated by the exponential averages \text{bAv} and \text{Bav}, which are calculated with the equations similar to (EQ. 10) and (EQ. 14):

\[
\begin{align*}
(16) \quad \text{dbAv/dt} &= -(\beta/\text{tauAv})*(\text{bAv} - b) \\
(17) \quad \text{dBav/dt} &= -(\beta/\text{tauAv})*(\text{Bav} - B)
\end{align*} \quad \text{(EQ. 16, EQ. 17)}
\]

Thus the short back-traffic bursts do not cause the change in the \text{tauAv}, but the long bursts, which might indicate the permanent change in the network configuration cause the \text{tauAv} to be equal to \text{tauRtt}. This causes the Q-algorithm to quickly converge to the intended back-traffic value, after which (when \text{bAv} becomes smaller than \text{Bav}) the regular large value of \text{tauAv} takes over and the short traffic bursts are effectively ignored - as they should be from the fractal traffic model standpoint.

So the final expression for the Q-algorithm averaging time \text{tauAv} is:

\[
(18) \quad \text{tauAv} = \max(\text{tauRtt}, \text{tauMax}), \quad \text{if} \quad \text{bAv} \leq \text{Bav} \quad \text{and} \quad \text{(EQ. 18)}
\]
\[ \tau_{Av} = \tau_{Rtt} \quad \text{if} \quad b_{Av} > B_{av}. \]

It is noted that even though the outgoing flow control block of the flow control algorithm is intended to be used only with the connections, which use the reliable transport at some level, the Q-algorithm has no such limitation. The Q-algorithm might be useful even when the connections do not have any reliable component whatsoever – for example, when the broadcast is implemented as the global radio (or multicast) broadcast to all receivers available. In this case the Q-algorithm would decrease the total network load by limiting the broadcast intensity to the level at which the majority of the responses can be processed by the network.

So far we have not discussed in detail the method of determining of the outgoing connection bandwidth used to route the back-traffic. As far as the Q-algorithm is concerned, the numerical value of \( B \) is just another input parameter provided to it by some other algorithm block. In case of the connections utilizing the reliable transport at some level in general and the Gnutella connections in particular, this value is determined by the fairness block of the flow control algorithm.

**Fairness Block**

The Fairness block is logically situated at the sending end of the connection and is a logical extension of the outgoing flow control block. Whereas the outgoing flow control block just makes sure that the outgoing connection is not overloaded and its latency is kept to a minimum, it is the job of the fairness block to make sure that: (i) the outgoing connection bandwidth available as a result of the outgoing flow control algorithm work is fairly distributed between the back-traffic (responses) intended for that connection and the forward-traffic (requests) from the other connections (the total output of their Q-algorithms); and (ii) the part of the outgoing bandwidth available for the forward-traffic broadcasts from other connections is fairly distributed between these connections.

The term fairly distributed as used here means that when some bandwidth is shared by the multiple logical streams, no stream or a group of streams should be able to monopolize the bandwidth, effectively starving or denying access by some other stream or a group of streams and making it impossible for them to send any meaningful amount of data.
Some mathematical notation, variables, and conventions are now introduced that will be used in subsequent description. Let $i$ be the connection number on the servant. For example, if the servant has five connections, $1 \leq i \leq 5$. This index is used to mark all the variables related to that connection:

$G_i$ – The total outgoing connection bandwidth.

$B_i$ – The part of the $G_i$ 'softly reserved' for the back-traffic. This means that when the back-traffic $b_i$ is less than or equal to $B_i$, it is unconditionally sent out without any delays.

$b_i$ – The outgoing back-traffic (responses) on that connection.

$f_i$ – The forward traffic sent from this connection to other connections to be broadcast. This is essentially equal to the output of the Q-algorithm $x$ described elsewhere herein.

$y_i$ – The total desired forward-traffic from the other connections – this is what they would like to be broadcast if there's enough bandwidth available. So $y_i = \text{sum}(f_j | j \neq i)$, plus the requests that are generated by this servant (not received by it to be broadcast).

$b_{oi}$ – The incoming response traffic intended for the other connections. $b_i = \text{sum}(b_{oj} | j \neq i) +$ the responses that are generated by this servant (not received by it to be routed) if the connection has enough bandwidth to route all the responses. Otherwise the $b_i$ value might fluctuate because of the response bufferization by the outgoing flow control block.

$f_{oi}$ – The outgoing request traffic sent out by this connection. It shares the outgoing connection bandwidth with $b_i$, so $f_{oi} = \min(y_i, G_i - b_i)$. This means that if the connection can send all the other connections' Q-algorithms outputs $f_j$, then $f_{oi} = y_i$; otherwise $f_{oi}$ is limited by the connection bandwidth available and the outgoing back-traffic $b_i$: $f_{oi} = G_i - b_i$. 
\( d_i \) — The part of the total desired forward-traffic \( y_i \) dropped by the algorithm when it cannot send all of it. \( d_i = y_i - \text{foi} = \max(0, y_i + b_i - G_i) \).

Thus the total outgoing connection bandwidth \( G_i \) is effectively broken into two sub-bands: \( B_i \) and \( G_i - B_i \), where \( 0 < B_i < G_i \) and \( 0 < G_i - B_i < G_i \). \( B_i \) is the part of the total bandwidth 'softly reserved' for the back-stream \( b_i \), and \( G_i - B_i \) — is the part 'softly reserved' for the forward-stream \( \text{f oi} \). (Here all the traffic streams do not include the 1-hop messages used by the outgoing flow control block to limit the connection latency — they are regarded as an invisible 'protocol overhead' by this block).

The term 'softly reserved' as used here means that when, for whatever reason, the corresponding stream does not use its part of the total bandwidth, the other stream can use it, if its own sub-band is not enough for it to be fully sent out. But if the stream \( b_i \) or \( y_i \) cannot be fully sent out, it is guaranteed to receive at least the part of the total outgoing bandwidth \( G_i \) which is 'softly reserved' for this stream regardless of the opposing stream bandwidth requirements. For brevity's sake, from now on, we will actually mean 'softly reserved' when we will apply the word 'reserved' to the bandwidth.

Let's imagine that we have some method of finding an optimal \( B_i \) and see how would the traffic streams behave themselves in that case. In the long run, the Q-algorithm would make sure that the mean value of \( b_i \) obeys the rule:

\[
\text{Eq. 19} \quad <b_i> \leq <B_i>/2, \quad \text{(since } \rho = 1/2) \quad \text{(Eq. 19)}
\]

Here \( <b_i> < <B_i>/2 \) if the incoming stream of requests on the connection is not powerful enough to fully occupy one half of the bandwidth \( B_i \) and \( <b_i> = <B_i>/2 \) otherwise.

This means that on the average at least one half of the total outgoing bandwidth \( G_i \) is available for the other-connections forward traffic \( \text{f oi} \). So on the average the mean value of the forward traffic from other connections \( <\text{f oi}> \) will be non-zero:

\[
\text{Eq. 20} \quad 0 < <\text{f oi}> \leq <G_i - b_i> 
\]
Now let's imagine that for some reason the back-traffic \( b \) experiences a burst which fully occupies the bandwidth \( B_i \), and at the same time the \( d_i > 0 \) (other connections wish to send more requests than what would fit into the bandwidth). The fairness requirement means that the forward-traffic streams, which have nothing to do with the response traffic burst, would not be greatly affected. In any case, this back-traffic burst should not bring the connection bandwidth dedicated to the forward-traffic to zero.

In fact we require that the back-traffic burst should not decrease the forward-traffic on the same connection by more than a factor of two. Then we can easily write the equation for the mean value of \( B_i \):

\[
(21) \quad <G_i - B_i> = \frac{1}{2}<f_{oi}> \quad \text{or} \quad <B_i> = <G_i - \frac{1}{2}<f_{oi}> \quad \text{(EQ. 21)}
\]

This equation can be strict only when the averaging interval is an infinite one. In reality, we need the mean value of \( B_i \) to converge to its ideal value:

\[
(22) \quad <B_i> \to <G_i - \frac{1}{2}<f_{oi} > = <G_i - \frac{1}{2}\min(y_i, G_i - b_i)> = <G_i - \frac{1}{2}(y_i - d_i)> \quad \text{(EQ. 22)}
\]

If the total outgoing connection bandwidth is known, we can immediately arrive from this to the differential equation for the \( B_i \) in the same fashion as we have arrived to the equation (EQ. 10) from (EQ. 9) in the Q-algorithm:

\[
(23) \quad dB_i/dt = -(\beta/\tau_{Av})*(B_i - G_i + \frac{1}{2}<f_{oi}>) \quad \text{.} \quad \text{(EQ. 23)}
\]

Unfortunately, in practice the value of \( B_i \) calculated with the help of that equation is essentially unusable – \( B_i \) is the slowly changing averaged value, and \( G_i \) is a quickly changing random input. So it is quite possible to have \( B_i > G_i \), which leaves no bandwidth whatsoever for the forward-traffic at that moment.

In order to solve this problem, the new variable \( r_i \) is introduced. \( r_i \) is the share of the total bandwidth \( G_i \) reserved for the back-traffic:
\[ (24) \quad Bi = ri \times Gi \quad \text{(EQ. 24)} \]

Then the expressions (22) and (23) can be written as:

\[ (25) \quad <ri> \rightarrow <1 - \frac{1}{2} * foi/Gi> \quad \text{(EQ. 25)} \]

and

\[ (26) \quad \frac{dri}{dt} = -(\beta/\tau_{Av}) \times (ri - 1 + \frac{1}{2} * foi/Gi) \quad \text{(EQ. 26)} \]

respectively. It is easy to see that \( ri \) calculated from the equation (26) cannot exceed 1. One can take the slowly changing value of \( ri \), multiply it by the current fast-changing value of \( Gi \) and arrive to the value of \( Bi \) to feed into the Q-algorithm which is guaranteed to be less than \( Gi \).

The variable \( ri \) also makes it simpler to implement the bandwidth sharing. To guarantee that as we do the send the back-traffic sending rate \( bi \) would not exceed the value of \( Bi \) defined as \( ri \times Gi \). Before we perform the actual sending operation, it might be very difficult to determine the precise value for \( Gi \). Even if we would know the exact link characteristics between the servant and its peers and would know the hardware bandwidth between them, there are several factors that can affect this value. First, the same hardware link is shared by several connections. Second, the outgoing flow control block significantly decreases the maximum theoretical throughput when it tries to minimize the connection latency. And third, there might be many other independent processes sharing the same hardware link – Web browsing, FTP transfers, and the like.

In fact, the only thing one may generally know about the connection is that the last time \( V \) bytes of data were sent, it took the outgoing flow control ‘echo PONG’ \( T \) seconds to arrive back. Theoretically it might be possible to create the data transfer model which would be able to predict the time \( T \) given the transfer amount \( V \). But in practice such models are very imprecise, noisy and generally unreliable. It is much easier to abandon the \( Gi \) prediction altogether and use \( ri \) to directly calculate the number of bytes in the total packet sent out when the ‘echo PONG’
arrives. If the total packet size is \( V \) (for example, \( V = 512 \) bytes when the outgoing flow control block has a large amount of data to send), we just have to allocate

\[
V_b = r_i^*V \quad \text{(EQ. 27)}
\]

bytes for the back-traffic data (responses) in this packet. Then regardless of what would be the value of \( T \) after which the echo will arrive, the resulting bandwidth value will be \( G_i = V/T \), and the \( B_i = r_i^*V/T \), which gives us exactly the value \( B_i = r_i^*G_i \) that we need.

Note that the \( B_i \) estimate for the Q-algorithm as \( B_i = Vb/T \) is not precise but the error has been modeled and shown not to be prohibitively large. In the worst case, when the estimate error reaches its maximum, the mean back-traffic \( <b_i> \) converges to about \( <0.707*B_i> \) instead of \( <0.5*B_i> \), and even this relatively small error can be observed only when \( <f_{oi} > - 0 \), which is not a very common situation for the real-life networks.

A second half of the fairness algorithm is now described, which assures the 'fair sharing' of the outgoing forward-stream \( f_{oi} \) between the request streams from the different connections. This algorithm block is very important in a practical sense, since it allows the coexistence of very-different-bandwidth connections on the same servant and is also largely responsible for the whole flow control algorithm ability to withstand the Denial-of-Service (DoS) attacks.

Unfortunately the full mathematical model of the servant behavior in case of the different-bandwidth links or a servant under the DoS attack is complex and is not described here in detail. In fact, an attempt to arrive to the analytical solution of the corresponding system of equations might be counterproductive. The reason for this is that this system contains about ten equations for every servant connection (the equations (EQ. 10 - EQ. 12), (EQ. 23), (EQ. 26) and some additional equations describing the interaction between the connections). The interpretation of the analytical solution even for five connections or so is prohibitively complex. This is why the servant behavior under these conditions was analytically modeled only in a steady-state approximation and numerically – for some simple cases (for example, the connections were divided into two groups, and all the connections in a group shared the similar characteristics).
Still, even in that case the full presentation of the results would take a lot of space, so this section of the document just presents the final results - the algorithm itself and some commonsense explanations of why it has been chosen.

Consider the servent A connected to the servent B and to some other servents C, D, E, and so on. Now consider the ‘fairness block’ of the connection, which links A to B and use index i to denote the variables related to that connection. At any given moment, if the summary flow of forward-traffic from other connections yi is less than or equal to Gi - bi, the bandwidth sharing is not an issue, since everything can be sent out anyway.

The problem of the fair forward bandwidth sharing arises only when yi > Gi - bi, where the value of bi is calculated by the forward/backward fair bandwidth sharing algorithm described above. Recall, yi was defined as yi = sum(f) | j != i, plus the requests that are generated by the servent A (not received by it to be broadcast). If yi > Gi - bi, some requests have to be dropped from the streams f in order to bring the total forward-traffic on the A-B connection to the value of foi = Gi - bi. The outgoing flow control algorithm block requires the requests to be prioritized according to their hop count, so the fair sharing algorithm should prioritize the requests from the different connections but having the same hop count. (One consequence of this approach is that the requests from the servent A itself will always have the highest priority if they are present, since these are the only requests with a hop count of zero.)

For a practical implementation, one may actually consider it to work in terms of the part Vf of the total packet volume V to be sent out. Vf is the part of the packet dedicated to the forward-traffic in a pretty much the same fashion as Vb was dedicated to backward-traffic in (EQ. 27). The reason for that approach would also be similar - the precise values of Gi, bi, etc are not known at the time of the packet sending. Still, here we will keep using the Gi, bi, foi variables for the illustrative purpose, even though in practice they are likely to be replaced by the variables V, Vb, Vf with little or no effect on the algorithm operation.

Some additional variables are now introduced, let fjk designate the part of the connection j incoming request stream which has the hop k and folk – to designate the part of the outgoing forward-traffic stream of the connection i, which carries the requests with a hop count value of k. The requests from the servent A itself are regarded as the requests with connection number j=0,
so they can be regarded as the regular part of the stream. The only special feature of this sub-stream with \( j = 0 \) is that it carries only the requests with a zero hop count, which does not affect the treatment of such requests by the fairness algorithm – they don’t receive any special priority. The fairness algorithm is supposed to be designed in a way which would send these requests first even without any knowledge about their importance. In the situation we are interested in, that is when the fair bandwidth-sharing algorithm has to be invoked, it follows that:

\[
\text{(28) } \sum \text{foik} \mid 0 \leq k \leq \text{maxTTL}) \ldots \leq \sum \text{sum(fjk} \mid j \neq i) \mid 0 \leq k \leq \text{maxTTL})
\]

Thus at least some of the requests in the streams \( \text{fjk} \) have to be dropped, and it is the job of the fairness algorithm to decide what is dropped and what is added to the outgoing streams \( \text{foik} \) to be sent out.

Before we describe the fairness algorithm in further detail, let’s consider a few examples to illustrate how it desirably should not work. Naturally, the fairness algorithm should not throttle down any connection’s stream to zero as it drops the requests. So, for example, it should not operate by taking all of the requests from connection 0, then 1, 2, … and dropping all the rest (connections m…N) when the bandwidth limit is reached. This approach would just arbitrarily choose the connections with high numbers and stop forwarding all traffic from them. (This hardly seems ‘fair’.) Similarly, it should not keep taking the random requests from the streams \( \text{fj} \mid j \neq i \) until the bandwidth limit \( \text{Gi} - \text{bi} \) is reached. Statistically that approach would seem to give every connection’s stream a non-zero bandwidth, but in practical terms, if some connection would generate a very high-rate request stream, its requests would occupy most of the bandwidth, leaving almost nothing for all the other connections. Specifically, if that high-rate stream would be a result of the DoS attack, the attacker would effectively reach its goal, preventing the servant A from broadcasting any requests than the ones generated by the attacker and bringing all the useful traffic through this host close to zero. The same objection is true for the method, which shares the outgoing stream \( \text{foi} \) between the request streams from other connections in proportion to their desired sending rate \( \text{fj} \), regardless of the method used to prioritize and to drop the different-hop requests \( \text{fjk} \) within the \( \text{fj} \) stream itself.
This is why the fairness block implements the ‘round-robin fairness algorithm’. This algorithm works as follows: first, for every connection with number \( j \neq i \), it prioritizes the requests it wishes to send according to their hop count values with small hop counts having the highest priority. This operation forms \( N \) logical queues, where \( N \) is the number of the servant \( A \) connections. (\( N-1 \) for the connections with number \( j \neq i \) and one queue for the ‘fake’ connection with number zero, which represents the servant \( A \) requests). Then it starts taking data from these logical queues’ heads in a round-robin fashion and transfers this data to the buffer to be sent out on connection \( i \). This transfer is being performed in a ‘hop-layer’ fashion, meaning that the requests with the hop count \( k+1 \) are not touched (the corresponding connections are just skipped) until all the requests with the hop count of \( k \) has been transferred. Thus regardless of how much data the connection \( j \) wishes to broadcast on the connection \( i \), when the send buffer is full (the \( Gi - bi \) bandwidth limit is reached), only a volume of data comparable to the other connections’ volumes will be actually sent.

Actually, the servant can perform a DoS attack and send a large amount of requests with a hop count of zero, effectively preventing its neighbors from broadcasting anything but these requests. But these neighbors will be able to send their own requests without any problems whatsoever, since the attacking requests will have a hop count of one and have the lower priority. The servants one more hop further from the attacking servant will be able to issue their own requests and to route the legitimate requests with a hop count of 1. The detailed examination of the traffic flows in such a case shows that such a DoS attack will weaken as the distance from the attacking host grows, and in no case will it significantly affect the user-perceived performance of the servants under attack.

It is desirable that this round-robin algorithm to transfer an equal number of bytes and not an equal number of requests from every logical queue. Otherwise, a DoS-attack is possible when the attacker issues very large (several kilobytes) requests, which effectively take over the whole outgoing forward-traffic bandwidth and prevent the useful requests from being sent.

One other way of implementing the round-robin fairness algorithm is to find the threshold values for the hop \( m \) and for the sending rate \( ft \). The hop is chosen so that all the request substreams \( fjk \) with hop value \( k < m \) fit into the \( Gi - bi \), and the request substreams with hop value \( k \leq m \) do not fit. The sending rate \( ft \) is calculated for sum of the ‘threshold hop’ \( m \)
request substreams \( \text{sum}(\text{fjm} | j \neq i) \). ft should allow the request substreams with the rate \( \text{fjm} \leq \text{ft} \) to be fully sent out, the streams with the rate \( \text{fjm} > \text{ft} \) to be sent out only at the rate \( \text{ft} \), and the total resulting stream to be equal to \( \text{Gi} - \text{bi} \):

\[
(29) \quad \text{Gi} - \text{bi} = \text{sum}(\text{foik} | k < m) + \text{sum}(\text{fjm} | \text{fjm} \leq \text{ft}) + \text{ft} \times n, \quad \text{(EQ. 29)}
\]

where \( n \) is the number of connections with \( \text{fjm} > \text{ft} \).

This way might be preferred since it does not require the full sorting of the request queues according to the hop count, but might be difficult to implement in case of the large-size requests. However, the implementation of the discrete-traffic (finite request size) fairness algorithm might be tricky regardless of which algorithm variant is used and is outside of this document scope in any case. For example, one might want to make sure that the round-robin operation does not start from the same connection every time to avoid the misbalance between the connections, but still might give a priority to the “A’s own” requests from the ‘pseudo-connection’ number zero.

Finally it is worth mentioning why the round-robin fairness algorithm works in a ‘hop-layer’ fashion rather than just taking the data from the connection queues regardless of their hop. The reason for this is that otherwise a connection with a low-rate input request stream would have even the high-hop requests broadcast, whereas the high-rate connection would have even the low-hop requests dropped. That would be unfair to the servants sending the requests through the network segments with the high-capacity links in the request routes. Such clients would have the ‘shorter reach’ - their ‘visibility radius’ would be lower, meaning that their requests would reach the lower number of servants than average. Since it was assumed to be wrong to punish the requestor for having the high-bandwidth link close to it in the network, the hop-layer round-robin procedure has been provided.

**Discrete Traffic Case**

Having now described embodiments of a flow control method and algorithm for a distributed broadcast route type networks in a continuous traffic approximation – it has been assumed that the traffic streams do not have a ‘minimum transfer unit’. Actually the requests and responses may not be infinitely small, and in fact can be very large. Note that Gnutella protocol specifies a limit of as high as 64 Kbytes for the request and the response size.
This fact has at least one noteworthy consequence for the practical implementation of the inventive method and algorithm. Some of these consequences were already mentioned, but the consequence of very large packet treatment deserves some additional amplification. The Gnutella protocol multiplexes several logical streams over a single TCP connection by sending the sequence of requests and responses belonging to the different substreams. This means that when a very large request or response is being pushed through the wire or other communication channel, nothing else can be transmitted over the same connection until it goes through. At the same time, the latency of all other connections on the same physical link goes significantly up even in the presence of the outgoing flow control block. This gives one an idea of the DoS attack utilizing the very large packets to bring the latency of the neighboring hosts to the unacceptable levels. Unfortunately, nothing can be done to fully deflect this attack until the Gnutella protocol is changed to use some other multiplexing method, which would allow the large requests to be sent out in small chunks.

In the meantime, the flow control algorithm described here should not broadcast or route the requests and responses bigger than some reasonable size (3 Kbytes or so). One might also try to close the connections to the servant, which try to send the messages of twice that size (5-6 Kbytes), since the latency on such connections would be too high for normal operation anyway, but such a behavior would open the door for the ‘old-servent DoS attack’. In this attack, the malicious host might try to broadcast the large requests through the ‘old’ servants, which are not aware of the flow control algorithm. Thus the attacker can hope to eventually reach many ‘new’, flow-controlled servants and remotely terminate all their connections to the ‘old’ ones. So in the absence of the proper low-latency stream multiplexing in the protocol it is preferable to just drop the big messages without closing the connection. Providing a smooth transition to the flow-controlled protocol without breaking the Gnutella network in the process may also be considered.

The foregoing descriptions of specific embodiments of the present invention have been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise forms disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, to thereby enable others skilled in the art to best utilize the invention and various embodiments with
various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the claims appended hereto and their equivalents. All publications and patent applications cited in this specification are herein incorporated by reference as if each individual publication or patent application were specifically and individually indicated to be incorporated by reference.
I Claim:

1. A method for controlling the flow of information in a distributed computing system, said method comprising:
   controlling the outgoing flow of information including requests and responses on a network connection to that no information is sent before previous portions of information are received to minimize connection latency;
   controlling the stream of requests arriving on the connection and arbitrating which of said arriving requests should be broadcast to neighbors; and
   controlling monopolization of the connection by any particular request/response information stream by multiplexing the competing streams according to some fairness allocation rules.
FIGURE 1

Each node may have one or more of outgoing flow control; Q-algorithm and fairness procedure, or may have none of these.
Appendix A. ‘Connection 0’ and request processing block.

This section does not introduce any new algorithms – it just describes the reasoning behind some architectural concepts presented in the previous sections of this document.

The GRouter block diagram (Fig. 1 in section 3 of this document) shows the ‘Connection 0’ block as the special ‘virtual’ connection that is used, among other things, to provide the ‘local’ responses to the incoming requests. It processes the requests to the servent and sends back the results (if any). The simplest example of the request is the Gnutella file search request – it initiates the search of the local file system or database and returns back the matching filenames (if found) as the search result. Of course, this is not an only imaginable example of the request – it is easy to extend the Gnutella protocol (or to create another one) to deliver the ‘general requests’, which might be used for many purposes other than the file searching.

The words ‘local file system or database’ do not necessarily mean that the file system or the database being searched is physically located on the same computer that runs the GRouter. It just means that as far as the other servents are concerned, the GRouter provides an access point to perform searches on that file system or database – the actual physical location of the storage is irrelevant. The ‘Connection 0’ block de-couples the request processing logic from the message routing one. This might be especially important when the local search API is implemented as a network API and its throughput cannot be considered infinite when compared to the TCP connections’ throughput. In that case it is clearly important to have the flow control logic in the local request-processing code in order to avoid overloading the request-processing engine.

Similarly, it is important to limit the number of locally processed requests when the uplink bandwidth is small, and an attempt to answer all the incoming requests might overload the outgoing communication channel.

The decision to make the interface to the local request processing block a regular GRouter connection that obeys all the connection flow control rules described in this document provides a simple and uniform method of handling the requests. It guarantees that regardless of the servent responsiveness to requests, its bandwidth limitations and the rate of the incoming requests, the response rate from the ‘local’ request-processing block won’t overload the servent outgoing bandwidth.

Further, one of the ways to implement the GRouter is to make it a ‘pure router’ – an application that has no user interface or request-processing capabilities of its own. Then it could use the regular Gnutella client running on the same machine (with a single connection to the GRouter) as an interface to
the user or to the local file system. The decision to handle the locally processed requests through the Connection 0 makes the servants functioning as the 'pure routers' (no Connection 0) logically identical to the servants that use the Connection 0 to access the local request-processing block. This fact gives a great deal of flexibility to the GRouter developer and provides a possibility to implement a wide array of local and remote request processing configurations, knowing that the flow control issues are guaranteed to be handled by the GRouter logic regardless of the chosen architectural solution.

However, at the first glance the 'Connection 0' algorithm seems to have one serious drawback. When the local request-processing interface (Connection 0) receives the same requests as all other connections, it is clear that only the requests that have been chosen by the Q-algorithms for broadcast (to be forwarded further on the network) have a chance of being answered locally. The consequence of this is that on the identical-node network the highest-hop requests might always be transmitted over the GNet with no effect and for no apparent reason. If, say, the Q-algorithms on all servants forward all the requests with hop 5 and drop all the requests with hop 6, then only the incoming requests with hop=5 will reach the Connection 0 and produce the responses. These requests will be also broadcast, and reach the peer servants as the requests with hop=6 – only to be dropped at once by the Q-algorithms. This seems illogical – due to the request multiplication these hop=6 requests would probably represent the majority of the request traffic, so why send such requests if there won’t be any responses anyway?

To answer that, first of all we need to introduce this identical-node network model in a more detailed fashion. Let us imagine ourselves the infinite network consisting of the identical servants having the same number of connections N+1, connected by the identical links and synchronously sending out the identical one-byte requests with the rate of one request per time step. Further, let’s imagine that every request forwarded to the peers by the servant is also locally answered by a Wp-byte response. The fact that the network is an infinite one allows us to view it as the network with no loops (redundant paths) and use an exponential broadcast multiplication rule as a result.

This ‘identical servant model’ is not very realistic, but can be a useful GNet analysis tool - it is very easy to analyze since the traffic on all connections is identical. It is convenient to write the stable traffic pattern for such a network as a table that shows how much data passes through every network link during a single time step and what is the layout of this data in terms of message hops and ttls. Such a table might look like the Table 1 below.

The hop value in the table starts with zero for the message that traverses its first link and is incremented every time the new link is traversed by the message. The ttl value is applicable only to the responses and shows how many total links have to be traversed by these responses to reach the destination (the requestor host). The link between the GRouter and its local request-processing block is not explicitly
shown anywhere in the table. It is assumed to be an infinite-bandwidth link to an infinite-speed request processor, so that the arrival of every request that is accepted for broadcast by the Q-algorithm immediately (on the next simulation step) causes the local response.

| Ttl  |  | Forward-traffic (requests) |  | Back-traffic (responses). (Wp units). |  |
|------|-----------------|---------------------------|-----------------|-------------------------------------|
| Hop  | *** not relevant*** | 1 | 2 | 3 | ... | 1 | 2 |
| Messages | 1 | N | N'(k-1) | N'k | ... | 1 | N | N'^2 | N'^2 | ... | 1 | N'k | N'k |
| Dropped by Q-alg. | 0 | 0 | 0 | 0 | ... | *** not relevant – full response transmission *** |
| Dropped by OFC | 0 | 0 | 0 | 0 | ... |

Table 1. The ‘identical servant model’ traffic layout example.

Note that thus defined hop and ttl values might be different from the actual protocol implementation binary values – for example, the Gnutella protocol defines ttl in a different and rather complicated way, and the presence of the local request-processing block would complicate the matters even more. Here we try to use the simple and obvious definitions for these table rows, because an attempt to bring the tables below in compliance with the Gnutella protocol binary fields’ meanings would make the issue very difficult to understand to anyone not intimately familiar with the Gnutella protocol binary specification.

The table consists of two parts – the forward-traffic part on the left shows the requests with different hop values that travel through each connection on every time step, and the back-traffic part on the right shows the responses with different ttl values that travel through the same connection. The number of per-hop request bytes grows with the hop value as a power of N. The number of response bytes grows in the similar fashion, but allows for the fact that the response to the k-hop request has ttl=k+1 and traverses k+1 network links before reaching its destination. This is why the responses with ttl value of k+1 form k+1 columns – the connection sees the same-ttl responses with different hop values that travel to the nodes separated from that connection by a different number of links.

Two bottom rows in the table represent the requests dropped by the servant Q-algorithm when there are too many responses and by the OFC block if there’s an excessive number of requests. The ‘Dropped by Q-algorithm’ row shows the requests that are being dropped upon arrival and is related to the
hop value of the arriving requests. On the contrary, 'Dropped by OFC' row shows the requests that are
dropped immediately before being sent, so their hop value is the one of the servant outgoing requests. Table
1 shows the unlimited request propagation case, and no requests are dropped, which explains zero values in
these two rows.

In the general case, the number of responses for each hop with \(\text{ttl}=k+1\) is equal to the number of
requests with \(\text{hop}=k\) minus the number of requests with this hop value dropped by the Q-algorithm. The
requests that are chosen for broadcast by the Q-algorithm are also forwarded to the infinite-bandwidth local
request-processing block and cause the responses to be sent back.

The Table 1 shows just the general table layout in case of the infinite request propagation (infinite
link bandwidth). Now let us present a similar table for the case when all the \(\text{hop}=k\) requests are forwarded
for broadcast by the Q-algorithm and all the requests with \(\text{hop}=k+1\) are dropped. The traffic layout in such
a case is shown in Table 2:

<table>
<thead>
<tr>
<th>Ttl</th>
<th>Forward-traffic (requests)</th>
<th>Back-traffic (responses). (Wp units).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*** not relevant ***</td>
<td></td>
</tr>
<tr>
<td>Hop</td>
<td>0 1 \ldots k-1 k k+1</td>
<td>0 0 1 0 1 2 \ldots 0 \ldots k</td>
</tr>
<tr>
<td>Messages</td>
<td>1 N \ldots N^(k-1) N^k N^(k+1)</td>
<td>1 N N N2 N2 N2 \ldots N^k \ldots N^k</td>
</tr>
<tr>
<td>Dropped by Q-alg.</td>
<td>0 0 \ldots 0 0 \ldots 0</td>
<td>*** not relevant – full response transmission ***</td>
</tr>
<tr>
<td>Dropped by OFC</td>
<td>0 0 \ldots 0 0 \ldots 0</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. The 'identical servant model' traffic layout in 'k-hop limit' case.

As it can be seen in Table 2, at some hop value \(k+1\) the infinite request multiplication can cease to
be possible. In this particular situation our original assumption was that it is the result of the Q-algorithm
operation due to the excessive amount of the response traffic, so Table 2 shows that all the \(\text{hop}=k\) requests
are locally answered and forwarded. When forwarded, the number of the requests is multiplied by \(N\) again
and appears in the \(\text{hop}=k+1\) forward traffic column as \(N^*(k+1)\). The cell below contains the number of
messages dropped on every time step by the Q-algorithm. This number is also equal to \(N^*(k+1)\), meaning
that these requests are dropped by the Q-algorithm immediately upon arrival.

"Dropped by OFC" line is empty (contains just zeroes), which is natural – this scenario assumes
that all the dropped requests are dropped by the Q-algorithm. This essentially means that the forward-traffic
bandwidth is not a limiting factor, even though with N=4 about 75% of the forward traffic is represented by the useless hop=k+1 requests. The requests are dropped by the Q-algorithm only, since the responses obviously have trouble fitting into the bandwidth reserved for the back-traffic (Wp is large enough). Section 7.1 introduces the constant relationship between the total link bandwidth and the bandwidth reserved for the back-traffic, so even if we would not send a single request with hop=k+1, it would not result in any response traffic increase.

This fact is important: let us imagine that somehow we would design an ‘ideal’ algorithm for request propagation, defining it as the algorithm that maximizes the individual request reach (the number of hosts that can answer that request). Regardless of this ‘ideal’ algorithm operation, it would not give us a better reach than the one we already have anyway – the response bandwidth is a limiting factor here, and no algorithm would allow us to receive more responses from any additional hosts.

Now let us consider an opposite situation, when the OFC block limits the request propagation. That is, the Q-algorithm allows us to forward every request (which also means that every arriving request is answered by the local request-processing block if it has an infinite processing bandwidth), but some of the requests are dropped by the “RR-algorithm & OFC block” (Fig. 2) from the request buffers.

This situation might arise when the value of Wp is low enough and the response traffic is not the limiting factor in the request propagation – the request reach is limited by the forward-traffic channel bandwidth.

Since the OFC block drops the requests in the hop-layered fashion, the traffic layout presented in Table 3 could describe that situation.

<table>
<thead>
<tr>
<th>TIL</th>
<th>Forward-traffic (requests)</th>
<th>Back-traffic (responses). (Wp units).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*** not relevant ***</td>
<td>1</td>
</tr>
<tr>
<td>Hop</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Messages</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>Dropped</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>by Q-alg.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dropped</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>by OFC</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3. The ‘OFC-limited’ case traffic layout.
Here the OFC algorithm detects the forward-traffic bandwidth overflow when \( x \) requests with \( \text{hop}=k \) per time step are being sent over the link. All the rest of the \( \text{hop}=k \) requests (\( N^k-x \) per time step) are dropped, as shown in the bottom line of the k-hop column. \( x \) requests per step reach the peers and since the Q-algorithm does not mind forwarding these for broadcast (by our assumptions, Q-algorithm forwards everything), cause the local request processing-block to immediately respond with \( W^p \times x \) replies. These replies are shown in the right (response) part of the table in \( k+1 \) different-hop copies as they travel across the GNet to the request source.

However, even though the Q-algorithm multiplies the k-hop requests and forwards them to other \( N \) connection blocks within the GRouter to be sent as \( \text{hop}=k+1 \) requests, these requests are dropped by the OFC algorithms within the connection blocks. Our initial assumption is that the OFC algorithms have trouble sending out even the \( \text{hop}=k \) requests, so the requests with the higher hop value have no chance to be sent out.

Comparing this situation with the ‘ideal’ broadcast algorithm again, we see that no algorithm could provide the wider request reach than the one we are using. The forward traffic channel is the limiting factor, and the only way to increase the request reach would be to get rid of some requests that are sent over the link but never answered, thus wasting the link bandwidth.

However, it is clear from Table 3 that every request that is sent over the link between servants does result in a response, and no bandwidth is wasted to transfer the requests that won’t have a corresponding reply. So our ‘Connection 0’ algorithm provides the best possible request reach again.

Finally, let us consider the general case, when we do not know in advance which traffic limitation algorithm is involved – maybe the Q-algorithm and the OFC-block can operate at once, each one dropping its share of requests. Indeed, this is something that is possible, and two cases should be considered:

In the first case it is the Q-algorithm that imposes the primary limit on the number of requests, meaning that the requests multiply freely until the Q-algorithm decides that the full broadcast of the \( N^k \) requests arriving with \( \text{hop}=k \) would result in an excessive response traffic.

The traffic layout in such a case is presented in Table 4 below. In Table 4 the column \( \text{hop}=k \) is the first column that has the request propagation limited. The Q-algorithm drops \( N^k-x \) arriving requests with \( \text{hop}=k \), passing just \( x \) requests further to be broadcast on \( N \) connections, which might result in up to \( N^k \times x \) requests with \( \text{hop}=k+1 \) on the next link. This is the maximal possible number of \( \text{hop}=k+1 \) requests. Now let us introduce the new variable \( F \) equal to the part of the bandwidth reserved for the forward-traffic still available after the requests with \( \text{hop} \leq k \) are sent. If \( F = N^k \times x \), all \( N^k \times x \) requests with \( \text{hop}=k+1 \) can be
sent, otherwise no more than F requests are sent and the rest are dropped by the OFC block that detects the bandwidth overflow.

<table>
<thead>
<tr>
<th></th>
<th>Forward-traffic (requests)</th>
<th>Back-traffic (responses). (Wp units).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ttl</td>
<td>*** not relevant ***</td>
<td>1 2 3 ... k+1</td>
</tr>
<tr>
<td>Hop</td>
<td>0 1 ... k-1 k k+1</td>
<td>0 0 1 0 1 2 ... 0 ... k</td>
</tr>
<tr>
<td>Messages</td>
<td>1 N ... (N^*k)-1 (N^*k)</td>
<td>1 N (N^{*2}) (N^{*2}) ... x x</td>
</tr>
<tr>
<td>Dropped</td>
<td>0 0 ... 0 (N^*k-x) 0</td>
<td>*** not relevant -- full response transmission ***</td>
</tr>
<tr>
<td>by Q-alg.</td>
<td>0 0 ... 0 0</td>
<td>max(0, (N^*x-F))</td>
</tr>
<tr>
<td>Dropped</td>
<td>0 0 ... 0 0</td>
<td>max(0, (N^*x-F))</td>
</tr>
</tbody>
</table>

Table 4. The general-case traffic layout with Q-algorithm as a primary limiting factor.

So the number of requests with \(\text{hop}=k+1\) is \(\min(N^*x, F)\), and the rest of these requests are going to be dropped by the OFC block (\(\max(0, N^*x-F)\) dropped requests). Regardless of the number of requests with \(\text{hop}=k+1\) on the link, none of these requests would be passed further to be broadcast by the Q-algorithm, since the Q-algorithm does not even broadcast all requests with \(\text{hop}=k\). Thus the local responses are going to be caused only by the forwarded \(\text{hop}=k\) requests (\(x\) total), as can be seen in the right side of the Table 4.

Now, since the Q-algorithm does limit the request broadcast, this means that the back-traffic channel is fully occupied by the responses (this is an only reason why the Q-algorithm would limit the broadcast). So we can see that regardless of the number of requests with \(\text{hop}=k+1\), no 'ideal' algorithm would widen the request reach -- the situation is similar to the one presented in Table 2.

In fact, if \(x\) in Table 4 is equal to zero, the Table 4 becomes effectively equivalent to Table 2 (just with a different \(k\) value).

The second case to consider is the OFC block being the primary limiting factor. This case is presented in Table 5. Here the \(N^*k\) requests to be sent with \(\text{hop}=k\) do not fit into the remaining forward bandwidth F. So \(N^*k-F\) requests are dropped and just F requests are sent over the link. When these \(\text{hop}=k\) requests are received, the Q-algorithm has to decide whether these requests should be forwarded for broadcast or not. Let's say that the Q-algorithm decides to forward x requests and drop all the rest (\(F-x\)). These forwarded requests result in the x local responses with \(\text{ttl}=k+1\) (as can be seen in the right part of the
Table 5. They might potentially result in $N^x$ requests with $\text{hop}=k+1$, but it does not happen, since the OFC block drops all of them - it cannot fully send even the requests with $\text{hop}=k$.

<table>
<thead>
<tr>
<th>Ttl</th>
<th>Forward-traffic (requests)</th>
<th>Back-traffic (responses). (Wp units).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*** not relevant ***</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Hop</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Messages</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$k+1$</td>
</tr>
<tr>
<td></td>
<td>Dropped by Q-alg.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>*** not relevant – full response transmission ***</td>
</tr>
</tbody>
</table>

Table 5. The general-case traffic layout with OFC block as a primary limiting factor.

If $F>x$, the Q-algorithm is limiting the request traffic, meaning that it is the response bandwidth that limits the individual request reach. So again, as in Tables 2 and 4, no ‘ideal’ algorithm would allow us to achieve the wider request reach than the one shown in Table 5 and achieved with the help of the ‘Connection 0’ (pure router) algorithm.

If $F=x$, the Table 5 becomes identical to the Table 3. Then the Q-algorithm does not drop any $\text{hop}=k$ requests upon arrival, so they are multiplied and would result in $N^x$ requests with $\text{hop}=k+1$, but since the forward bandwidth is not big enough to send these, they are all dropped. This case has been already analyzed previously and it has been shown that it also provides the widest possible request reach.

So this limited-scope modeling shows that the seeming drawbacks of the ‘Connection 0’ algorithm do not actually result in the decreased individual request reach when such an algorithm is used. The only disadvantage of this algorithm is the excessive request traffic when the response volume is high and the Q-algorithm is used to limit the request propagation. This excessive request traffic, however, does occupy only the bandwidth reserved for the request traffic anyway, and in many practically important cases might be even nonexistent. For example, the typical Gnutella file-searching servant has the low response volume and is likely to use mostly the OFC algorithm to limit the request propagation. In that case the ‘extra’ request traffic present in Tables 2, 4 and 5 would not be present at all and the traffic layout would be similar to the one presented in Table 3.
At the same time the simplicity of the 'Connection 0' or 'pure router' interface to the local request-processing block makes it very valuable from the implementation standpoint. Since some form of the flow control is necessary in this interface in any case, the alternatives to the 'Connection 0' algorithm are likely to be highly complicated. For example, a separate Q-algorithm or its analog would be necessary to assure that the local responses won't overload the outgoing servent links. This separate Q-algorithm would have to interact somehow with the 'normal' Q-algorithms in order to achieve the fair bandwidth sharing between the local and the remote responses. The precise sharing layout would have to depend on the numerous factors -- the local interface bandwidth, the probability of the local response and so on, complicating the matters even more.

So the additional request data sent over the bandwidth that is likely to be left unused anyway is a small price to pay for the architectural flexibility and the simplicity of the servent implementation.
Appendix B. Q-algorithm step size and numerical integration.

This section describes the practical approach to the Q-algorithm (equations (50-56)) result computation. This process includes the numerical integration of several differential equations (50), (53-55) and even though it is pretty straightforward from the computational mathematics standpoint, it is sufficiently complicated to deserve an explicit description.

The ultimate goal of the Q-algorithm is to determine the share \( x(t) \) of the incoming request traffic \( f(t) \) that should be forwarded to the other connection blocks for the further broadcast. In practical terms that means that for every new incoming request of size \( V_{req} \) (the one with the GUID that was not encountered before) its desired number of bytes to broadcast is calculated and then the request is passed to the other connections. This desired number of bytes to broadcast is calculated as \( V_{ef} = V_{req} \times x(t)/f(t) \) – the details of this procedure were described in sections 8.1 and 8.2.2 (equation (84)).

So the value of \( x(t) \) has to be known at the moments when the network packets arrive on the connection. That makes the intervals between the network packet arrivals the natural choice for the Q-algorithm step size \( T_q \) in time domain. When the network packet arrives, the Q-algorithm input parameters \( B, b, f, R \) and \( tau_{Rtt} \) (see section 8.2) have to be calculated. \( R \) and \( tau_{Rtt} \) are calculated according to the algorithms described in sections 8.2.1 and 8.2.2. \( b \) and \( f \) are the actual observed incoming rates for the data arriving on the connection. \( b \) is the total volume of responses that have arrived from the other connections to the response prioritization block (Fig. 2) during the Q-algorithm step time \( T_q \), divided by \( T_q \). \( f \) is the total volume of the requests that have arrived to the Q-algorithm block from the duplicate GUID rejection block (Fig. 2) during the time \( T_q \), divided by \( T_q \).

\( B \) is the outgoing response bandwidth estimate. According to (26) (section 7.1), it is found as \( 2/3 \) of the outgoing connection bandwidth estimate, calculated from (13,14) (section 6.2). Unlike the values for \( f \) and \( b \), this estimate is performed every time the OFC PONG echo returns for the outgoing packet. So the new value for \( B \) becomes available not when the requests are received, but when the OFC PONG is received, which might happen at different time moments. Thus regardless of the Q-algorithm time step \( T_q \), the equation (55) that calculates \( B_{av} \) from \( B \), works in its own time scale (with different step size \( T_b \)) to find the averaged value of \( B_{av} \) from \( B \). The latest available value of \( tau_{av} \) is used by the equation (55) in the process.

After all these Q-algorithm input variables (\( B, b, f, R \) and \( tau_{Rtt} \)) are computed, we can make the Q-algorithm step. This step can be described as calculating the ‘new’ values (at time \( t \)) for the output variables (\( Q, u, x, Rav, B_{av}, b_{av} \) and \( tau_{av} \)), using the ‘old’ (at time \( t-T_q \)) values of the output variables.
and the 'new' values of the input variables. Since the equations (50-56) are dependent on each other and several equations can use the same variable (for example, $\tau_{av}$ is used in (50) and (53-55)), the order of the computation might be important from the numerical stability standpoint. Generally we want to use the latest available variable values in any particular equation.

Setting aside the $b_{av}$ variable that is calculated in a different time scale, and the final Q-algorithm output $x$, which is calculated from the equation (52) as $\min(f, Q/R_{av})$ when the 'new' values for $Q$ and $R_{av}$ are known, we can concentrate on the subset of the output variables. This subset includes $Q$, $u$, $R_{av}$, $b_{av}$ and $\tau_{av}$. In this subset $Q$ depends on $u$ and $\tau_{av}$, $u$ depends on $Q$ and $R_{av}$, $R_{av}$ depends on $\tau_{av}$, $b_{av}$ depends on $\tau_{av}$, and $\tau_{av}$ depends on $b_{av}$. As we start doing the step, we have the old values for all these variable from the previous step output or from the initial conditions.

First, we can determine the new value of $b_{av}$ from (54), using the old value of $\tau_{av}$ (and the input variable $b$). This new $b_{av}$ value (together with the latest $b_{av}$ value) allows us to arrive to the new $\tau_{av}$ value, using (56).

The new value of $\tau_{av}$ makes it possible to find the new value for $R_{av}$ from (53), using also the input value $R$. The new value of $R_{av}$, in turn, makes it possible to find the new value for $u$ from (51) (using an input variable $f$ and the 'old' value of $Q$), and that operation allows us to calculate the new $Q$ value from (50). Finally, the equation (52) can be used to calculate the new value for $x$.

Note that the order described above is not an only way to calculate the new output variable values. For example, we might start with calculating the new $\tau_{av}$ value from (56) using the old $b_{av}$ value and the latest $b_{av}$ value, and only after that use (55) to compute the new $b_{av}$. This approach, however, would result in the latest response data $b$ being effectively unused in the $\tau_{av}$ calculation. It would allow us to use the same new $\tau_{av}$ (averaging time) value in all differential equations (50) and (53-54), but it would also result in a slower Q-algorithm reaction to the response bursts. This averaging time value would be only 'half new' – it would not reflect the latest responses, and for this reason such an approach is not desirable. Even though the Q-algorithm was specifically designed to avoid the quick control actions and to change the request broadcast rate $x(t)$ slowly and gradually in order to increase the algorithm stability, it is better not to introduce the additional instability (like this 'half-old' $\tau_{av}$ variable) into the algorithm if possible.

The intentionally slow reaction speed of the Q-algorithm also makes it possible to choose a variety of other time steps instead of the one between the two network packet arrival times described above. The developer might find the amount of computations necessary when such a time step is used to be excessive, since the broadcast rate $x(t)$ is likely to change very slowly. So the developer might choose the bigger time
step for the Q-algorithm, using the latest computed value of \( x(t) \) to determine the request broadcast rate in the meantime.

To a certain extent, such a step size increase is certainly possible – it is just important to be careful and to remember that all the information received during this bigger time step has to be used in the Q-algorithm computations. For example, \( x(t) \) should not exceed \( f(t) \) (as can be seen from (52)). So if the bigger step time is used, \( f(t) \) should be also computed from the data arriving to the connection block over that bigger time. No request can have the desired number of bytes to broadcast \( (V_{req} \times x(t))/f(t) \) bigger than the request size \( V_{req} \), so the ratio \( x(t)/f(t) \) should never exceed 1. Similar considerations apply to all the other input variables – for instance, the estimated response channel bandwidth \( B \) used in (50) should not be just a latest sample of this variable, but should be the averaged value of the bandwidth estimates observed during the step time.

Regardless of the Q-algorithm step size choice, it is important to remember that the Q-algorithm contains four differential equations - (50), (53), (54) and (55). The similar differential equation (49) controls the Q-block of the RR-algorithm. All these equations are essentially the ‘averaging’ ones – they are used to compute the exponentially averaged values of the quickly changing input variables. The quickly changing character of the input data makes it necessary to be careful when performing the calculation ‘step’ to find the new value of the output variable.

Let us consider the model differential equation

\[
(86) \quad \frac{dy(t)}{dt} = -\frac{1}{\tau} (y(t) - z(t)).
\]

This equation is similar to the equations used by the Q-algorithm and we can use it to illustrate the possible approaches to averaging of the quickly changing function \( z(t) \). Let us use the index \( i \) to designate the variable values at time \( t - dt \) and \( i + 1 \) – to designate the values at time \( t \), at the end of the calculation step with time \( dt \). The simplest way to compute \( y_{i+1} \) is

\[
(87) \quad y_{i+1} = y_i - \left( \frac{dt}{\tau} \right) (y_i - z_{i+1}) = (1 - \frac{dt}{\tau}) y_i + \left( \frac{dt}{\tau} \right) z_{i+1}.
\]

The first problem associated with this solution is that it is numerically unstable when the large time step \( dt \) is used. For example, if we set \( dt = 3 \tau \) and \( z(t) = 0 \), it is easy to see that (87) becomes

\[
(88) \quad y_{i+1} = 2 y_i.
\]
that is, instead of converging to zero, \( y(t) \) starts to oscillate with an increasing magnitude. This problem can be avoided if the time step \( dt \) is decreased or the ‘forward-looking’ form of (87) is used:

\[
y_{i+1} = y_i - (dt/\tau) \cdot (y_{i+1} - z_{i+1}).
\]  

(89)

Finding \( y_{i+1} \) from (89), we can write it as

\[
y_{i+1} = (y_i + (dt/\tau) \cdot z_{i+1}) / (1 + dt/\tau).
\]  

(90)

This solution is numerically stable regardless of the \( dt \) value, though, of course, its precision drops rapidly as \( dt \) grows. For example, setting \( z(t) = 0 \) again, we can compare the precise solution

\[
y_{i+1} = y_i \cdot \exp(-dt/\tau)
\]  

(91)

and the numerical solution

\[
y_{i+1} = y_i / (1 + dt/\tau).
\]  

(92)

It is easy to see that \( dt=0.1*\tau \) results in the 0.5% error for \( y_{i+1} \). \( dt=\tau \) gives us the 36% error, and when \( dt=3*\tau \), the error grows to 400%.

Thus regardless of whether the equation (87) or (90) is used, the excessive values for \( dt \) are not desirable in any case.

Another problem associated with the numerical solution (87) of the equation (86) is the ‘overshoot problem’. Due to the fractal nature of the network traffic the functions averaged by the Q-algorithm normally have a very high variance – in other words, these functions are very ‘bursty’ and can have very high (potentially unlimited) peaks. On the intuitive level it is clear that since the equation (86) produces an ‘averaged’ value of the input function \( z(t) \), its output function \( y(t) \) should converge to the \( z_{i+1} \) value as the time step \( dt \) grows. However, since the equation (86) contains the \( (dt/\tau) \cdot z_{i+1} \) component, its result \( y_{i+1} \) can ‘overshoot’ the target value of \( z_{i+1} \) as the step size \( dt \) grows to \( dt=\tau \).

This is a purely numerical effect that is closely related to the already discussed issue of the solution (87) numerical stability. In fact, it is easy to see that the same approach (90) used to counter the numerical instability of the equation (87) also solves the ‘overshoot problem’. As the time step \( dt \) grows to \( dt=\tau \), the \( y_{i+1} \) calculated from (90) does converge to \( z_{i+1} \). Of course, this numerical convergence is not an exponential one dictated by (86) - \( y_{i+1} \) changes as \( z_{i+1}/(1+\tau/dt) \) as \( dt \) grows. Even though this
convergence is qualitatively correct and cannot, for example, lead to numerical overflows (as it is the case with (84)), the convergence rate is still wrong, which underscores the need to choose the time step \( dt < \tau \).

If the Q-algorithm averaging time \( \tau \alpha V \) is very low (or the intervals between the request packets arrival \( T_q \) are very high), it might so happen that \( T_q > \tau \alpha V \) (or \( dt > \tau \)) in terms of the equation (86)). Then the step interval \( T_q \) has to be broken into several smaller sub-steps \( dt < T_q \), and the integration should be performed in several steps. Every sub-step should use the same value of \( z(t) = z_{th} \) (whatever that is in the context of the particular Q-algorithm differential equation) and the newly calculated (on the last sub-step) value of the output function \( y(t) \). Of course, if the input function \( z(t) \) is available in several points inside the integration interval \([t-T_q, t]\), these values can be used instead of the constant value \( z(t) = z_{th} \) that would be the same for the whole integration interval. That would increase the integration precision, but typically won't be necessary unless the integration interval \( T_q \) is too long or these multiple values for \( z(t) \) would be available anyway and it won't matter whether to use the single or the multiple values for \( z(t) \) from the CPU load standpoint.

The next consideration to have in mind is the relationship between the equations (50) and (51). The equation (50) calculates the new value for \( Q \), using (among other variables) the latest known value for the 'estimated underload' factor \( u \), which, in turn, depends on \( Q \) through (51).

That relationship makes the choice of the time step \( dt \) especially important for the equation (50), since as the value of \( Q \) passes through the \( f^*Rav \) barrier, the shape of the function \( u \) and the right side of the equation (50) change dramatically. If \( u > 0 \) \((Q > f^*Rav)\), the right side of the equation (50) becomes proportional to \((\rho^*B - f^*Rav)\) and no longer depends on \( Q \). Effectively the equation (50) can be treated as two different equations depending on the value of \( Q \):

\[
\begin{align*}
(93) \quad dQ/dt &= -(\beta/\tau) V Q (Q - \rho^*B) ; \quad Q \leq B_{av}, \quad \text{if} \quad Q \leq f^*Rav, \quad \text{and} \\
(94) \quad dQ/dt &= -(\beta/\tau) V (f^*Rav - \rho^*B) ; \quad Q \leq B_{av}, \quad \text{if} \quad Q > f^*Rav.
\end{align*}
\]

The equation (94) is not the equation of the same type as the equation (93). The solution of the equation (94) is a simple linear function of the form \( Q(t) = A + t + C \), so all the considerations presented above for the equation (86) do not apply, and any time step \( dt \) can be used in that equation as long as the original assumption \( Q > f^*Rav \) remains valid. However, since the behavior of the equations (93) and (94) might be very different, we have to keep track of whether \( Q \) does pass through the \( f^*Rav \) barrier during the Q-algorithm step \( T_q \), and change the equation, if necessary, in the middle of the Q-algorithm step.

One of these equations ((93) or (94)) should be chosen at the beginning of the time step \( dt = T_q \) according to the old value of \( Q = Q_0 \), and to the new values of \( f \) and \( Rav \). After that, the equation similar to
(87) or (90) can be used to check whether the value of \( Q_{lu} \) would deviate sufficiently from \( Q_i \) to make the original choice of the equation (93) or (94) invalid at the end of the interval \( T_q \). If that happens to be the case, the interval \( T_q \) should be broken into two at the point \( t \) where the original equation choice becomes invalid, and the separate equations ((93) or (94)) should be used in both parts of the interval \( T_q \).

Since all the variables in the equations (93) and (94) except \( Q \) (that is, \( \beta, \rho, \tau, \alpha, B, f \) and \( \text{Rav} \)) are effectively constant on the interval \([t-T_q,t]\), four scenarios are possible:

- \( Q_i < \text{f}^{*}\text{Rav} \) and \( Q_i > \rho^*B \). In this case \( Q_{lu} < Q_i \), and the equation (93) is used on the whole time interval \([t-T_q,t]\).

- \( Q_i < \text{f}^{*}\text{Rav} \) and \( Q_i < \rho^*B \). Then the equation (93) is used in the starting point of the interval \([t-T_q,t]\) and \( Q \) starts to grow. If \( Q_{lu} > \text{f}^{*}\text{Rav} \) when the time step \( dt=T_q \) is used, it means that somewhere in the middle of the step \( T_q \) the equation (93) has to be replaced by the equation (94). Then also \( \text{f}^{*}\text{Rav} < \rho^*B \) — otherwise \( Q \) would never reach the \( \text{f}^{*}\text{Rav} \), since it would not exceed \( \rho^*B \). Thus the equation (94) causes \( Q \) to keep growing, and this time point when the equation (93) had to be replaced by (94) is a single equation-replacement point on the interval \( T_q \). So the equation (94) can be used up to the end of the interval (unless \( Q \) exceeds \( B_{av} \), in which case the \( Q \leq B_{av} \) limit applies).

- \( Q_i > \text{f}^{*}\text{Rav} \) and \( \text{f}^{*}\text{Rav} < \rho^*B \). In this case \( Q_{lu} \geq Q_i \) (unless it is already limited by the condition \( Q \leq B_{av} \)) and the equation (94) is used on the whole time interval \([t-T_q,t]\).

- \( Q_i > \text{f}^{*}\text{Rav} \) and \( \text{f}^{*}\text{Rav} > \rho^*B \). Here the equation (94) is used in the starting point of the interval \([t-T_q,t]\) and \( Q \) starts to decrease. If \( Q_{lu} < \text{f}^{*}\text{Rav} \) when the time step \( dt=T_q \) is used, it means that somewhere in the middle of the step \( T_q \) the equation (94) has to be replaced by the equation (93). Then \( Q = \text{f}^{*}\text{Rav} > \rho^*B \), and the value of \( Q \) would keep decreasing, so again this equation-replacement point is a single such point on the interval \([t-T_q,t]\).

In other words, in no case the interval \([t-T_q,t]\) should be broken into more than two intervals, and no equation-replacement procedure can result in the \( Q(t) \) changing direction on this interval.

There is another way to achieve the same goal. When \( Q \) is changing towards \( \text{f}^{*}\text{Rav} \), we can perform the numerical integration of the equation (50) with step \( T_q \) as a sequence of smaller steps \( dt \), each of which would result in just a small change in the value of \( Q \) (for example, no more than \( 0.1^*\text{f}^{*}\text{Rav} \)). All the \( Q \)-algorithm input parameters should remain the same on every sub-step within one step, but the value of \( u \) should be recalculated at the end of the each sub-step according to (51). Since the equations (93) and
(94) are equivalent when $Q = f^*Rav$, the potential ~10% imprecision at the moment when the equations are replaced should not cause any major numerical errors.

The requirement $dQ = 0.1f^*Rav$ leads to the following sub-step $dt$ limit for the equation (50):

From equations (50) and (87) we can see that this requirement is equivalent to the following sub-step $dt$ size limit when $u=0$ (which is equivalent to using the equation (93) for this sub-step):

$$dt <= 0.1f^*Rav*tauAv / (beta*(rho*B - Q)), \quad \text{if} \quad u=0 \text{ and } Q < rho*B.$$  \hspace{1cm} (95)

If the equation (90) is going to be used for the numerical integration of (50), this condition would be a little different:

$$dt <= 0.1f^*Rav*tauAv / (beta*((rho*B - Q) - 0.1f^*Rav)), \quad \text{if} \quad u=0 \text{ and } Q < rho*B.$$  \hspace{1cm} (96)

The expression (96) is valid only when $(rho*B - Q) > 0.1f^*Rav$. Otherwise, no sub-step time can possibly result in the excessive $Q$ value change ($dQ$ never exceeds $0.1f^*Rav$), so any step size can be used. The additional condition $Q > rho*B$ in both (95) and (96) is used to limit the sub-step size only if $Q$ grows and can potentially exceed $f^*Rav$; if this is not so, any step size $dt$ can be used.

Naturally, regardless of whether the equation (87) and condition (95) or equation (90) and condition (96) are used, the usual step size conditions still apply. $dt$ should be less, and preferably much less, than $tauAv$.

When $u > 0$, which is equivalent to using the equation (94) for this sub-step, from (50) and (51) we can arrive to the following requirement for the sub-step $dt$ size limit:

$$dt <= (Q-f^*Rav)*tauAv / (beta*(f^*Rav - rho*B)), \quad \text{if} \quad u > 0 \text{ and } f^*Rav > rho*B.$$  \hspace{1cm} (97)

Here $Q(t)$ is a linear function and no other conditions apply to the $dt$ sub-step size. The additional condition $f^*Rav > rho*B$ is used to limit the step size only if $Q$ decreases and can potentially drop below $f^*Rav$; if this is not so, any step size $dt$ can be used. $Q - f^*Rav$ is used instead of $0.1f^*Rav$ in (97) since $Q(t)$ is linear and the value of $Q = f^*Rav$ can be reached in a single sub-step (if the step size $Tq$ is big enough to do it on this Q-algorithm step at all). Otherwise, small values of $f^*Rav$ might result in the very small sub-step size $dt$ and lead to the excessive number of computations necessary to make just a single Q-algorithm step $Tq$.  

133
The expressions (95-97) can become formally undefined when the denominators in these expressions are zero, or can lead to the $dt=0$ when $f, Rav$ or $tau Av$ are equal to zero. However, generally such situations arise when the same equation ((93) or (94)) has to (or can) be used during the whole interval $[t-Tq,t]$ anyway. For example, $f^{\bullet}Rav=\rho B$ when $u>0$ means that $Q_{0}=Q_{m}$. If $f^{\bullet}Rav=0$, we can assume that $u>0$ and effectively use the equation (94) on the whole interval $[t-Tq,t]$. And as it was already mentioned, when $u=0, \rho B-Q<=0.1f^{\bullet}Rav$ and the equation (50) is integrated with the method (90), no sub-step time can possibly result in the excessive $Q$ value change. Thus even if the change of the equation is required somewhere inside the interval $[t-Tq,t]$, we can still keep using the same equation and it won’t cause a major numerical error.

So whenever such a problem arises and the time step value $dt$ formally becomes zero or undefined, we can safely assume that any value of $dt$ will suit our needs (of course, within the normal numerical stability- and precision-related step size limits). Actually very small $dt$ values might be more dangerous than zero or undefined ones. The arbitrarily small sub-steps might result in a very high computational volume required to do just a single $Q$-algorithm step $Tq$ – this is why the additional conditions are introduced in the expressions (95-97). These additional conditions ($Q<\rho B$ for (95,96) and $f^{\bullet}Rav>\rho B$ for (97)) limit the applicability of the corresponding expressions and impose the time step limit $dt$ on the calculations only when $Q$ is moving towards $f^{\bullet}Rav$ boundary and can potentially cross it. In addition, the $Q-f^{\bullet}Rav$ multiplier in (97) is used to make sure that if $u>0$ just a single sub-step would be required to make a single $Q$-algorithm step $Tq$ or to cross the $Q=f^{\bullet}Rav$ boundary (whichever happens first).

This multi-step approach to the integration of the equation (50) is more computationally intensive than the explicit integration interval $[t-Tq,t]$ division and multiple-case analysis suggested earlier. However, it is simpler to implement and can be preferable when the load that is placed on the CPU by this method is not excessive.

All other differential equations ((53-55) and (49) in the $Q$-block of the RR-algorithm) do not require such a complicated approach to the choice of the time step size. It is enough to keep the value of $\beta * dt/tau <=0.1...0.4$, which would keep the numerical error within about 0.5%...10% range and allow to use the simpler equation (87) instead of (90). Since in the practical $Q$-algorithm implementation the recommended value of $\beta$ is 1, this requirement can be translated into

$$dt <= 0.1...0.4*tau Av.$$

If this requirement cannot be satisfied when $dt=Tq$, the integration should be performed in several smaller steps, each of which should satisfy (98).
A few words should be said about the Q-algorithm input value $B$ – the response bandwidth estimate. As it was already mentioned earlier in this section, the individual estimates of $B$ are computed when the OFC PONGs arrive on the connection. This does not necessarily coincide with the arrival of the requests on the same connection, so effectively $B$ is computed and averaged (by the equation (55)) in its own time scale. Of course, the usual integration step size limitation (98) applies to the equation (55).

The resulting $B_{av}$ value is used in the equation (56) to determine $\tau_{Av}$. The $B_{av}$ is a slowly changing variable, so it does not matter that it is calculated in a time scale that is different from the time scale used to calculate the other Q-algorithm variables.

However, the quickly changing variable $B$ is directly used in the equation (50), so some special measures should be taken to assure that the right value of $B$ is used there. Let us consider the model situation when two or more OFC PONGs arrive during the single Q-algorithm step $T_q$. If just the latest available value of $B$ would be used in (50), all other $B$ values would be unused, which would result in the increased Q-algorithm error. So if more than one estimate of $B$ is performed during the Q-algorithm step time $T_q$, some averaged $B$ value should be used in (50) in order to fully use all the available information about the response channel bandwidth.

Theoretically, the ‘normal’ averaged value $B_{av}$ could be used; however, this is not desirable, since it would effectively double the Q-algorithm reaction delay $\tau_{Av}$. Besides, such a long-term averaging with a characteristic time $\tau_{Av}$ is not functionally necessary here anyway – the equation (50) already provides the long-term averaging. In fact, the equation (50) was specifically designed to average the quickly changing input data $B$. What we need here is not a long-term averaging to smoothen the rapidly changing function $B(t)$, but rather a very short-term averaging with a characteristic time $T_q$. This short-term averaging is used to combine all the available information about the response bandwidth during the time $T_q$ into a single variable $B_{oh}$ to be used in the numerical solution of the equation (50).

The situation is complicated by the fact that the exponential averaging with an averaging time $T_q$ (Q-algorithm step size) cannot be used. The value of $T_q$ is normally defined by the interval between the arrival of the network packets with requests and is not known in advance. So the simple averaging is more appropriate here. Further, it is logical to use the weighted averaging, giving more weight to the values of $B$ calculated after the bigger time delay $\delta t$ (time that has passed since the last OFC PONG arrival and the last $B$ estimate).

This choice of weighting is determined by the physical nature of the data-sending process. Consider the following sequence of alternating $B$ estimates: $B=2$ KBytes/sec after 1000 ms delay, then $B=1$
KByte/sec after 500 ms delay and so on. The straight averaging would give us 1.5 Kbytes/sec bandwidth estimate. However, the 2 KBytes/sec value is valid longer than the 1 KByte/sec one. From the data-sending standpoint, during the 1500-ms period we should be able to send 2 KBytes during the first 1000 ms, and 0.5 KBytes during the remaining 500-ms interval. So the total bandwidth would be \((2\text{KBytes}+0.5\text{KBytes})/1.5\text{sec}=1.67\text{KBytes/sec}\). This estimate is equivalent to the weighted averaging method suggested above.

Using the index \(m\) to mark the individual \(B\) estimates and the corresponding delays during the \(Q\)-algorithm step time \(T_q\), we can write this averaging method as

\[
B_{i+1} = \frac{\text{sum}(B_m*deltat_m)/\text{sum}(deltat_m)}{\text{sum}(deltat_m)}/\text{sum}(deltat_m),
\]

where \(B_{i+1}\) is the averaged value of the response bandwidth estimate to be used in the numerical solution of the equation (50) to find the value of \(Q_{i+1}\) on the \(Q\)-algorithm step number \(i+1\).

This averaging method does not have the excessive storage requirements even when many OFC PONGs arrive during the single \(Q\)-algorithm step. We just have to remember two sums (for \(B_m*deltat_m\) and for \(deltat_m\)), the latest values of the same variables and the arrival time of the latest OFC PONG. When the connection is opened, the process is started with the zero values for all these variables. As soon as the first OFC PONG arrives, we have our first non-zero bandwidth estimate. Of course, there's no interval \(deltat\) associated with it, so we should just use this single \(B_m\) value as the \(B_{i+1}\) in the meantime. The second OFC PONG brings the first \(deltat\) value, which should be also used to weight the first estimate (so the first average is going to be just the average between the two \(B_m\) values). All the subsequent OFC PONGs can use (99) in the normal fashion.

When the requests arrive on the connection, the equation (99) is used unless the sums in it are equal to zero – if this happens, the saved \(B_{i+1}\) value from the last \(Q\)-algorithm step is to be used. (Naturally, before the first OFC PONGs arrive this value is equal to zero too). After the \(Q\)-algorithm step is performed, the \(B_{i+1}\) that was used in it is saved to be used if no OFC PONGs would arrive during the next \(Q\)-algorithm step. At the same time both sums in (99) are zeroed. If some OFC PONGs do arrive during the next \(Q\)-algorithm step interval \(T_q\), the sums are increased; otherwise the saved \(B_{i+1}\) value will be used.

Note that the first \(deltat_m\) value to be used during the \(Q\)-algorithm step \(T_q\) might be related to the bandwidth estimate interval that starts before the \(Q\)-algorithm step interval \([t-T_q, t]\); thus it is entirely possible to have \(\text{sum}(deltat_m)>T_q\). This approach allows us to fully cover the timeline \(t\) with the response bandwidth estimates and results in the correct bandwidth \(B\) averaging for the equation (50). Logically this
averaging approach for $B$ is similar to the technique used in section 8.2.1 to average $R(t)$ over the Q-algorithm step size $T_q$.

Finally it should be noted that all the equations presented in this section assume that the variables are the floating-point real numbers, and all the computations are performed as the floating-point operations. Normally the modern CPUs have the fast floating-point arithmetic and this approach should not present any problems. If, however, the GRouter code is implemented on the hardware where this approach results in a low performance, the equations can be easily written in the integer or fixed-point arithmetic terms. This operation, however, requires some careful analysis of the operational ranges and required precision of the variables and is outside the scope of this document.
Appendix C. OFC GUID layout and operation.

This section describes the possible allocation of the bits in the Globally Unique Message Identifier (GUID) of the OFC messages – the PINGs and PONGs that are used by the Outgoing Flow Control block. The standard Gnutella 128-bit GUID is used as an example, though the same approach to the GUID data layout can be easily applied to the other similar broadcast-route networks.

The message header in the Gnutella protocol contains the 128-bit (16-byte) GUID that is used for various purposes – to drop the incoming requests if these requests were already received earlier, to route back the replies and so on. The protocol does not clearly define the method to create the GUID – it just has to be sufficiently random to avoid having two messages with the same GUID. In practice, some implementations use the Windows function CoCreateGuid(), some use the pseudo-random number generator – the specific method is not important as long as the returned result is really unique.

The 128-bit size of the GUID provides a very high degree of ‘uniqueness’ that is not really necessary in practice. The message lifetime is finite and for all practical purposes the GUID can be considered ‘unique enough’ if it has a very low probability of collision with all the other GUIDs it can be possibly compared with during its lifetime. This probability can be estimated as the probability of meeting the same GUID value in the routing tables of all the servants that the messages with this GUID can possibly reach. In the typical Gnutella environment the request can reach about 10,000 servants and each of these servants can have the routing table with about 10,000 entries. So it can be potentially compared with up to 10^8 other GUIDs. In fact, this number is much lower since the routing tables on different servants have many identical entries, but let us use the 10^8 number as the upper boundary for the number of the other GUIDs that the GUID can be compared with.

10^8 is approximately equal to 2^27, so with the k-bit GUID the probability of the collision for the individual GUID is going to be about 2^(27-k). If the servant issues one request per second for 10 years, it generates about 2^28 GUIDs, so the probability of the individual servant ever (once in 10 years) having its GUID confused with some other GUID is about 2^(55-k).

If the GNet has 10^12 (that is, 2^40) servants (several hundred servants for every human being on the Earth), the probability of any one of these servants even having the collision-related problem during the 10-year interval is about 2^(95-k). For a k=128 bit GUID this amounts to 2^(-33), or about 10^(-10). By any standards, this is a very low probability, especially if we remember that nothing really devastating happens when the GUID collision does occur. Any hacker with an idea of mounting the attack on the Gnutella network can effectively simulate the GUID collision on his servant, so the GNet has to deal with
this problem anyway, whether it would be the result of the statistical GUID collision or the result of the hacker attack.

All these considerations have prompted various groups in the Gnutella community to come up with various suggestions that would effectively limit the GUID size with a smaller number of random bits and use some parts of the GUID for various GNet infrastructure-related purposes.

For example, the LimeWire proposal (Lime Wire, 18 West 21st Street, Suite 901, New York, NY 10010) suggests using the bytes 8 and 15 of the GUID (assuming the byte numbering from 0 to 15) for the protocol versioning. This proposal effectively leaves 112 ‘random’ bits in the GUID, which is still more than enough for any conceivable purpose.

In one aspect, LimeWire™ Versions 1.0 and 1.3 are commercially available (free) software package purported to enable individuals to search for and share computer files with anyone on the Internet. A product of Lime Wire, LLC, this LimeWire™ software is purported to be compatible with the Gnutella file-sharing protocol and can connect with anyone else running Gnutella-compatible software. At startup, the LimeWire program will connect, via the internet, to the LimeWire Gateway, a specialized intelligent Gnutella router, at startup to maximize the user's viewable network space. Limewire is written in Java, and will run on any machine with an Internet connection and (at the time of this writing) the ability to run Java version 1.1.8.

The OFC block does a similar thing. It has to mark the OFC PING with such a GUID that when the OFC PONG returns from the peer, this GUID could be recognized as an answer to the OFC PING and the appropriate action (network packet sending, bandwidth estimate calculation, etc) could be performed.

The OFC packets have ttl=1 and are supposed to travel only between two connected peers. The host that sends the OFC PING and receives the OFC PONG in response makes all the OFC decisions. The host that receives the OFC PING does not analyze its GUID or undertake any actions on the basis of this PING except for the normal PONG-sending operation. Thus the OFC GUID layout does not have to adhere to any GNet-wide standards - if the host is able to recognize its own OFC GUIDs that are returned back to it in OFC PONGs, it is quite enough for the Outgoing Flow Control algorithm purposes.

Still, the OFC GUID layout and the algorithms that work with it have to satisfy several basic requirements:

- The OFC algorithms should not rely on the PING being correctly interpreted by the peer. Even though the PING with the ttl value of 1 is not supposed to be forwarded to any third
host, some buggy and/or protocol non-compliant servents are likely to do so. The GUID layout and handling procedures should be prepared for that possibility. For example, one PING might result in several PONGs, the same PING or PONG can appear on the different connection and so on.

- Even though the peer is supposed to answer to the 1-ttl PING with the PONG, it is likely that for a variety of reasons certain percentage of PINGs won’t result in the corresponding PONGs. So if the PONG does not arrive after a certain reasonable timeout (~1 sec), the PING should be retransmitted. However, if the peer is just overloaded and both PINGs would eventually result in PONGs, the ‘additional’ PONGs should be discarded by the OFC block.

- The OFC GUID should be reasonably unique. Even though it has a very low travel distance (just one hop) and thus has the far smaller collision chance than the normal request, it still should have enough unique bits to avoid the collision.

- If the collision does happen, it should not result in any disastrous effects for the servent.

- The algorithms should be resistant to the malicious peers that might try to influence the OFC decisions by deliberately changing the GUID in the OFC PONGs.

Of course, there are many possible ways to generate the OFC GUID layouts that satisfy these conditions. Perhaps the simplest one is to generate the normal GUIDs and to remember these in the separate OFC ‘routing’ tables (one table per connection). These tables would not be actually used to route anything – just the GUID-searching functionality of the normal routing tables should be implemented in order to recognize the incoming 0-hop PONGs as the OFC messages. When the connection OFC GUID table is not empty (the connection is waiting for the OFC PONG), the connection also has to store the OFC packet information – the sending time and the packet size for the last OFC packet sent out.

This information is necessary for the OFC block to perform the outgoing bandwidth estimate as defined in the section 6.2 (equations (13) and (14)) and section 7.3 (equations (45) and (46)). According to the sections 6.1 and 6.2, if the OFC packet (the block of data between the two OFC PINGs) is sent as several TCP/IP (network) packets, we have to remember the whole OFC packet size, not the sizes of the individual network packets. This is the size of the OFC packet ‘payload’ – the ‘trailer’ OFC PING size is not included into it and is treated as the OFC overhead – we are interested in the bandwidth available for the regular request/response messages. The OFC packet sending time (when the sending is done with several subsequent networking calls) should be the starting time of the first call. Even though the GRouter is supposed to make these calls immediately one after another, this sequence of calls can still take some
noticeable time, and we are interested in the roundtrip time between the whole sending operation start time and the OFC PONG arrival time.

Normally there’s just one GUID in such a ‘routing’ table. The additional entries might be added when the OFC PING is being re-sent because of the PONG loss or excessive (more than ~1 second or so) PONG delay. This operation does not change the last OFC packet size and sending time stored in the connection block. Since the retransmitted PINGs are treated as an overhead, the lost or delayed PONGs just increase the roundtrip time for the same payload, effectively lowering the connection bandwidth estimate.

In fact it might be even possible to use the same GUID for the retransmitted OFC PINGs. However, this approach is not recommended, since it is desirable for the OFC algorithms to be independent of the specific peer implementation. Using the same GUID for the retransmitted PINGs might lead to the OFC deadlock. If the peer is implemented in such a way that it can lose the first OFC PONG as a result of the networking call error or for some other reason and then just keep rejecting the retransmitted PINGs with the same GUID, the OFC PONG might never arrive from it.

As the PONG with hop=0 and ttl=1 arrives on the connection, its GUID should be checked against this OFC routing table. If the match is found, all the necessary calculations should be performed - the bandwidth estimate should be updated, the OFC packet with messages should be sent out (including the new OFC PING at the packet end) and so on. Before the new OFC PING is sent, all the GUIDs should be removed from the OFC routing table. This is necessary to avoid the similar reaction to the possible delayed or duplicate PONGs with the same GUID, to the PONGs that come in response to the other (re-sent) PINGs, and to the delayed PONGs that come in response to the PINGs that have been sent earlier.

This ‘normal GUID’ approach is certainly usable. However, we have to perform the GUID searches in the table just to answer a simple question: is this PONG one of the PONGs that we are waiting for? Given the fact that the table in question probably never has more than 10-20 entries (the servant is likely to time out the connection as unresponsive after that), the GUID table search seems to be an excessively complicated solution for such a simple task.

So the alternative method is to effectively break the GUID bits into two groups: the randomized connection identifier GUID-C and the PING sequence number SEQ. The PING sequence number is some number, which would allow us to uniquely identify the OFC PING when the OFC PONG is received in response. For example, this number can be set to zero for the first OFC PING sent on the connection and then incremented for every subsequent OFC PING. Naturally, sooner or later this number would overflow and become zero again, so it should have enough bits for the delayed PONGs to be out of SEQ range regardless of the delay and still leave enough GUID bits for the connection identifier GUID-C.
For example, if the full GUID has 128 bits, 16 of which are used for the version tracking, we have 112 bits for the GUID-C and SEQ. The rollover time of the SEQ is determined by the lowest possible interval between the outgoing OFC packets and by the biggest possible PONG delay in case of the buggy peer implementation. If, say, the GNet is deployed on the 100-Mbits/sec LAN and 300-byte packets are used, the interval between the outgoing packets can be as low as 25 microseconds. On the other hand, the worst-case PONG delay can be as high as several hundred seconds. The ratio between these numbers can be as high as $10^{17}$, or $2^{23}$, so the SEQ number should have at least 24 bits, leaving 88 bits for the GUID-C. This value is still quite large and the collision probability is within the acceptable limits. In fact, in the practical implementation it might be even possible to allocate 32 GUID bits for the SEQ. That would allow the GRouter code to avoid the problems related to the 24<->32 bit internal conversion and alignment, and the remaining 80 bits for the GUID-C would be enough to avoid the GUID collisions.

When the GUID contains the OFC PING SEQ number, every OFC PING sending operation updates the acceptable range of the incoming PONG SEQ numbers. This range is associated with the connection, and both ends of this range are set to the PING SEQ value as the trailer OFC PING is sent after the OFC packet. If no PONGs are received in response, the subsequent OFC PING packets are sent with the incremented SEQ values and the acceptable range is also extended. When the OFC PONG finally does arrive with the GUID-C part of the GUID being the same as the connection’s GUID-C and the PONG SEQ number within the acceptable range, the necessary OFC operations are performed. After that the acceptable SEQ range is cleared and the procedure is repeated.

Of course, the presence of the SEQ information in the GUID makes it possible for the malicious peer to ‘spooﬁng’ the OFC algorithm. The malicious host can receive the OFC PING, modify the GUID and send the PONG with the modiﬁed GUID back. However, it is hard to imagine why the malicious host would engage in this activity. There’s not much sense in modifying the GUID-C, since such PONGs would be just dropped by the OFC block, which won’t recognize these packets as the valid OFC PONGs and would eventually close the connection as ‘unresponsive’. The same thing would happen if the malicious host would modify the SEQ numbers in a way that would place these numbers outside the acceptable SEQ range. If the SEQ number is modified to be still within the acceptable range, the only result achieved by the attacker might be the lowered bandwidth estimate. This can only decrease the bandwidth available for any DoS attack undertaken by the attacker, which makes such a GUID ‘spoofing’ attempt useless.

Another way to ‘spoof’ the GUIDs would be to predict the future SEQ numbers and send the OFC PONGs before the corresponding PINGs are received. If the malicious host performs such an operation accurately, it can create an impression of the ‘inﬁnite’ bandwidth between the attacked host and the attacker.
This attack makes more sense, since it can cause the attacked host to fully broadcast the requests from the attacker, which is something that could have been prevented by the attacked host Q-algorithm if it would have the correct bandwidth estimate. However, this approach would not allow the attacker to do anything more harmful than the 'normal' DoS attack that would be fought by the fair bandwidth sharing algorithms and the Q-algorithms of the GNet servents in the usual fashion. Furthermore, this erroneous bandwidth estimate might quickly overload the link with messages, increasing the link latency, causing the TCP overloads and eventually resulting in the connection shutdown and the attack termination.
1. A method for controlling the flow of information in a distributed computing system, said method comprising:
   - controlling the outgoing flow of information including requests and responses on a network connection to that no information is sent before previous portions of information are received to minimize connection latency;
   - controlling the stream of requests arriving on the connection and arbitrating which of said arriving requests should be broadcast to neighbors; and
   - controlling monopolization of the connection by any particular request/response information stream by multiplexing the competing streams according to some fairness allocation rules.

2. A method for assuring that the response flow does not overload the connection outgoing bandwidth in a communication system.

3. A computer-readable medium whose contents cause a computing device to perform the method of claim 1.

4. A computer system comprising components capable of performing the method of claim 1.
Figure 2
Figure 3

Figure 4
Figure 6
Hop-layered request buffer

Figure 8
Figure 10
Figure 11
Figure 14

Figure 15