INTERNET IMPROVEMENT

The following hola patents and patent applications are referenced by this patent and incorporated in it in whole: Patent #8135912 (SYSTEM AND METHOD FOR INCREASING CACHE SIZE), patent application number #12836059 (SYSTEM AND METHOD FOR PROVIDING FASTER AND MORE EFFICIENT DATA COMMUNICATION), patent application number #13034239 (METHOD AND SYSTEM FOR INCREASING SPEED OF DOMAIN NAME SYSTEM RESOLUTION WITHIN A COMPUTING DEVICE).

TUNNELS

A different way to implement the acceleration system described in hola's patent application 12836059 can be the following:

Figure 50 describes an alternative system that works as described in Hola patent 12836059.

In the network of figure 50, there is a new type of element called a Tunnel. As in patent 12836059, each of the communication devices in the network may store the same software that under different conditions enables that communication device to assume a different role, and may also take multiple roles at the same time (e.g. the communication device can be both a Client and a Tunnel for example), and thus a device may for example communicate with itself under different roles (for example a Client may use itself also as a Tunnel).

Figure 55 describes how an element takes on one of the roles that it can assume: 5506, 5508 and 5510 show that when the system is starting up, a connection is established with the Acceleration Server 1, and a list of Tunnels for this communication device is requested. Once received, in 5510 the device establishes connections (e.g. TCP connect) to each of the Tunnels in the list, so that once messages are needed to be sent to/from these Tunnels, it can be done quickly due to this pre-connection.

Block 5514 shows that upon receiving a protocol message from one of the other network elements in this network, the software in the device determines which role it should assume – i.e. Client, Agent, Peer or Tunnel, and in 5516 calls the software module within the device that assumes that role.

In 5520, part of the software acts as a proxy between the applications and the communication stacks, in such a way that when an application sends a request to the network, this is intercepted by the software, and sent to the Client portion of the software, for the software to handle this data as a Client in the network.

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 1 of 85

Figure 56 is the data structure that is used by all of the network elements when storing information about the data in the network, similar to that which is described in patent 12836059.

Figure 58 is a diagram of how Tunnels can be used by a Client. The "Tunnel" 16 is used by the Client 2 to fetch data from the Data Server 5, when such data is not found in the acceleration network's elements Caches, or the data is found in the caches but does not make sense for the Client 2 to fetch it for practical reasons, such as because the time to fetch it from the acceleration network's caches would be longer than to fetch the data directly from the Data Server 5. In such a case where it is beneficial for the Client 2 to fetch the data directly from the Data Server 5, the Client 2 may use one or more Tunnels 16 to carry out the request to the Data Server 5 on its behalf.

This approach has multiple benefits, including the ability to fetch multiple segments of this data in parallel (which may result in a faster load time) and also by creating multiple paths to the Data Server 5 (which may result in a more effective use of the network's resources).

Therefore, once the Client 2 has determined that it is not worthwhile to use the caches within the various acceleration systems' network elements, it will determine how many Tunnels 16 to use for this request based on the amount of data to fetch, and the speed of the available Tunnels 16. The Client 2 will also determine how much of the data it will request from each of the Tunnels 16 based on their speed, and thus assign a 'range' of the data (for example for a data of 1200 bytes where two agents A and B are selected, a range assignment may be that A is assigned the bytes between byte 0 and byte 856, and B is assigned bytes 857 to 1200). The Client 2 will then send the request to all these Tunnels 16 along with the requested range for them to fetch from the Data Server.

Claims:

1. **A network element of type "Tunnel" is added to patent 12836059, where the Tunnels receives a data request from a Client, including the data server to get the request from and the associated other data required for such a data request, and fetches this from the data server and provides back to the Client**
2. **The system described in 1, where a range is also provided to the Tunnel, and the Tunnel only fetches this range of data from the data server.**
3. **The system described in 1, wherein a Client chooses one or more Tunnels and a range per Tunnel, sends the request to these Tunnels, and receives back the information from the Tunnel that the Tunnel received from the Data Server**

Choosing whether to access the data directly (through Tunnels) or through the acceleration system

Often it is possible for a Client 2 to fetch data from the data server through Peers 4 (for example if they have cached this data before), and it is always possible for the Client 2 to fetch the information through Tunnels 16 by requesting that they fetch the information on the Client's behalf from the Data Server 5.

It is therefore beneficial for the Client to choose the option which will eventually lead to shorter load times of the data.

CLAIM:

1. **In an acceleration system with Peers and Tunnels, where the information required may be provided from the Peers as well as by the Tunnels, a system for determining which path to use by evaluating the time to receive the information from either path, by using the data from previous interactions with Peers and Tunnels in the vicinity of the available Peers and Tunnels, and using that path**

Figures 61 and 62 are flowcharts that suggest an implementation of a method of choosing the optimal method – whether to fetch the data from the acceleration network (called in the figures "PATH_ACCEL"), or directly via the Tunnels (called in the figures "PATH_DIRECT").

In 6102, the application's network request is intercepted by the acceleration software. In 6103 it checks whether the data required is located in the local cache of this communication device, and if it is, this is the data returned to the requiring application in 6130. If the data does not exist in the local cache, then the Client in 6104 estimates whether it is preferable to make the request in PATH_DIRECT or PATH_ACCEL. If there is not enough data to determine what is best, or if both paths are estimated to produce similar results, EQUAL is chosen. For determining whether to use PATH_DIRECT, PATH_ACCEL or EQUAL, multiple algorithms may be implemented, including some that are further described herein.

The algorithm that determines which path to choose, will assign an estimated time for retrieving the data in both scenarios. In 6106 the parameter T_DIRECT is set to the time that the algorithm assumes it will take to get the data loaded using the Tunnels, and likewise T_ACCEL is set to the time that the algorithm assumes it will take to get the data loaded by using the Peers.

The flowchart continues in Fig. 62. The system has now chosen whether to go through Tunnels (PATH_DIRECT), or Peers (PATH_ACCEL) or both (EQUAL). If PATH_DIRECT or PATH_ACCEL are chosen, it is beneficial for the system to set a timer (TIMER) after which time if the data is not

received, the alternative path should be tried. For this purpose, if PATH_DIRECT is chosen, 6110 sets TIMER to T_DIRECT plus a certain buffer (in this example 20% is used), and in 6112 the PATH_DIRECT (i.e. working with the Tunnels) is started, until either the request is fully completed, or TIMER expires (i.e. TIMER seconds have passed for example). In a similar manner, if PATH_ACCEL is chosen, 6114 sets TIMER to T_ACCEL plus a certain buffer (in this example 20% is used), and in 6116 the PATH_ACCEL (i.e. working normally with the acceleration system, i.e. using the Acceleration Server, Agents and Peers) is started, until either the request is fully completed, or TIMER expires (i.e. TIMER seconds have passed for example).

In both cases (i.e. choosing PATH_DIRECT or PATH_ACCEL), 6118 evaluates the result of this path. If TIMER has expired (i.e. the path did not yield the expected result before TIMER expired), then 6120 calls for both paths (DIRECT and ACCEL) to be done concurrently. Once one of the paths starts yielding the first piece of information (e.g. the first byte), the other path is terminated in order to save bandwidth. In other implementations this may be modified. For example, so that the other path is terminated only once all the data is received from the other path.

6123 waits for all information to be received from one of the paths. When a path has provided all the requested information, store all the meta data learned in this data request, including the data about the network elements that provided information, whether they succeeded or not, the time to connect to them at the TCP level, DNS times, time to first byte received, time to last byte received, etc. The storing of this information is done so that the software can later estimate better which path is better to pursue for future requests.

In order to keep the network up to date about which network elements have which data stored on them, and about the response times of each element, 6126 sends the meta data about the Clients 2, Peers 3, agents 4 and tunnels 16 to all network elements that participated in this transaction.

6128 sends the resulting data (received directly from the Data Server 5 through the Tunnels 16, or from the acceleration network through the Agents 4 and Peers 3, to the requesting Application.

CLAIM:

1. **the system in claim 1, where once the desired path is chosen, a timer is set for the expected time that this chosen path should yield the result, and if that time plus a certain buffer have passed, both paths are chosen at once**
2. **the system in claim 2, where upon receiving the first piece of data from a chosen path, if there is another chosen path that is active, that other path is terminated**

3. **the system in claim 2, where upon receiving the last piece of data from a chosen path, if there is another chosen path that is active, that other path is terminated**
4. **the system in claim 1, where upon getting back all the data to fulfill the request, information about all of the participating network elements, including response times for one or more of their functions, is stored for future use**
5. **the system in claim 5, where the information about all participating network elements is also sent to other elements in the network for future use**

Figure 63 suggests implementations for how the Acceleration Server 1 allocates network elements to act as Tunnels 16 for Clients 2.

A first suggested implementation 6302 is that once the acceleration server receives a request (6302) from a Client for Tunnels, the acceleration server returns **the IP of the client itself (i.e. the client can use itself as the tunnel, or combinations with other tunnels)** to the Client.

A second suggested implementation 6308 is that once the acceleration server receives a request (6302) from a Client for Tunnels, the acceleration server creates a list **of the 5 network elements that are the closest (in terms of IP or GeoIP or distance in terms of network speed) to the Client. For example, the acceleration server may use the speed stats engine algorithm described** in Figure 64 to determine which network elements have the fastest communication speed with the Client, and to choose them as the Tunnels. The acceleration server 1 then sends 6314 the tunnel list back to the requesting Client.

CLAIM:

1. **in a system where one or more network elements (tunnels) are used to load portions of data on behalf of another network element (a client), a method of choosing tunnels for the clients, where the network elements that are the closest to the Client are chosen as tunnels**
2. **the system in claim 1, where 5 such elements are chosen**
3. **the system in claim 1, where the closeness of the network elements to the client is determined by the GeoIP distance between them**
4. **the system in claim 1, where the closeness of the network elements to the client is determined by the IP distance between them**
5. **the system in claim 1, where the closeness of the network elements to the client is determined by the physical distance between them**
6. **the system in claim 1, where the closeness of the network elements to the client is determined by the data communication speed between them**

Fig. 64 suggests a method to test whether the PATH_DIRECT, PATH_ACCEL or EQUAL should be chosen. In 6202 the information about the request is received. 6403 looks for the information

that the system gathered (in previous transactions) about the participating network elements in the past, including their DNS lookup times, the TCPIP response times, their internal response to queries, their success rates at providing information to this Client, etc. 6404 checks whether such information exists. If for one or more elements this information does not exist (for example, possibly because that network element has never communicated with this Client), then 6406 checks whether this information may be estimated, based for example on previous interactions with network elements that are close (in geography or IP) to that network element for which information is missing. If no information exists, and no information can be estimated in 6406, then in 6408 EQUAL is returned (i.e. to use both the DIRECT and ACCEL paths).

In 6410 the total time to complete this request through PATH_DIRECT (T_DIRECT) and through PATH_ACCEL (T_ACCEL) is estimated based on the information fetched in 6403 and 6406. This may be done in a multitude of ways that would be obvious to one skilled in the art, for example by adding up the times of all of the actions that will need to be performed to complete the interaction. When adding the various times of the process (including DNS lookup, TCP connect, bandwidth to the network element, bandwidth from the element, etc.), there will typically be more than one data point for each such times of the process (for example, if the Client connected to this network element 10 times in the past, there will probably be 10 different TCP_connect samples. For the purpose of the calculation, use the number which 95% of the samples fall under (can use other numbers instead of 95%). 6413 shows that for the evaluation of the interaction time with the various network elements, it is required to also check the expected time it would take to transmit the payload to the element (i.e. payload_to_element/bandwidth to the element) plus the time it would take for the network element to transmit the result back to the Client (i.e. payload_from_element/bandwidth from the element).

6414 compares T_DIRECT to T_ACCEL to see if it more beneficial to use one path over the other. Because the two times are only heuristic assessments, it is beneficial to use some kind of 'watermark' when comparing. In the flowchart, a 20% watermark is used, so that if PATH_DIRECT is 20% faster than PATH_ACCEL, then it is used (6416), and vice versa if PATH_ACCEL is 20% faster (6420). If one of the paths is less than 20% faster than the other, then EQUAL is returned (6418) – i.e. both paths should be used.

Claim:

1. **a system for evaluating two or more paths of fetching data, where historical data for each stage of the connection on each path is referenced and added up, so that the complete time of one path may be compared to the second path**

2. **the system of claim 1, where the times used for each stage are the top percentile under which most samples fall (e.g. use the sample which is larger than 95% of the other samples)**
3. **the system of claim 1, where a watermark system is used to determine a threshold over which to prefer one path over another**
4. **the system of claim 3, where if neither paths are over the threshold, both are used**

Figure 69 describes the PATH_DIRECT – i.e. how to retrieve the data from the Tunnels 16 when this option is chosen.

In 6901 the Client 2 chooses which of the Tunnels 16 that were allocated to it by the Acceleration Server 1 it will use for this current transaction. Choosing all of the tunnels is an obvious way to implement 6901. In 6902, the requested data is split in to ranges, such that there is one range per participating Tunnel. It is possible to simply choose ranges of equal size to implement this. A possibly more efficient implementation is to provide each Tunnel a range that is proportional to that Tunnel's ability to provide the data (for example, based on that Tunnel's upstream and downstream traffic to the Client, as measured in previous communications between them).

In 6904 the data request is sent to each of the tunnels along with the segment that this Tunnel needs to fetch from the Data Server. The Tunnel will then make that request to the Data Server and return the information received to the Client.

In 6906 the timer is set for each Tunnel, where the timer is the max time that is allocated to this Tunnel to provide back the information to the Client. One way to choose the length of this timer, is to set it to the T_DIRECT that was defined in Fig. 64. Another way is to set the timer to the average response times of that Tunnel based on past experiences for example.

In 6910 it is evaluated whether all responses have been received, and if they are the function ends. If not all responses have been received, 6912 checks whether any timers expired on tunnels, and if they have, then an alternative Tunnel is chosen and the request is sent to that Tunnel instead. An alternative tunnel can be chosen by several methods; a random tunnel from the list of Tunnels for this client, or a Tunnel that is currently not in use, or a Tunnel that the Client has the highest BW/RTT ratio to, etc.

Figure 70 shows an example of how a tunnel may be implemented. The Tunnel receives a request (7002) from another network element (e.g. a Client) which includes the standard information of a data request (DR) including the address of the data server (such as the name or IP of a web server), the information requested (such as a URL), the range of the data that is requested (e.g. all of the data, or only a certain byte range, for example bytes 1000 to 1020), and additional information about the request (such as the cookies in HTTP). The Tunnel the

creates a request (7004) to the data server based on the DR and sends it to the data server. Once the response is received (7006), the Tunnel sends the data (7010) back to the requesting Client.

Peer Maps

In Hola patent 12836059, a Peer may provide data to a Client if the Peer has all of the data for the file requested by the Client. This is shown in Figure 72, in block 7202, where the information for a file is stored as an entry with the file identifier 7204 as a unique identifier of the file that is requested, and a list of Peers (7206, 7208, 7210) that are known to have cached this file that is identified by the file identifier 7204.

Often Peers will either have all of the file within their cache, or none of it. Sometimes however, Peers have only loaded and cached part of the file, or cached the file fully but discarded parts of it (for example because of limited storage space) and thus have only a part of the file stored.

For these cases it is beneficial to also provide Peers that have only part of the file stored, so that more sources may be used per request.

Figure 74 is a flowchart that shows how a peer map is created. First, in 7402 the file to be fetched is split up in to ranges, so that there is one peer per range (out of the peers that are available for that range). Ranges without peers are marked as 'other'. In 7404, a range request is sent for each range to the peer allocated to it in the peer map. In 7406 if the peer did not answer the range request in full (or did not answer at all after a specified amount of time), send that range request to other peers that have the information of that range. There are cases where there is no one peer that has the whole range, and in that case use multiple peers for getting the range. If there are parts of that range that there are no peers for, mark them as 'other'. In 7408 all ranges marked as 'other' are retrieved as range requests directly from the server that has the content.

CLAIM:

1. **in a caching system where network elements are used as sources for data that they've cached in the past for Clients, the ability to use network elements as Peers even if they have only a portion of the file cached, by storing information on which portion of the file is located on which element, and loading the information from the various elements**
2. **the system of claim 1, where the caching system is hola's patent 12836059**

3. **the system of claim 1, where when multiple Peers are available for a certain segments, the Peer who's BW/RTT is the highest among them is the one to be chosen to provide that segment**
4. **the system of claim 3, where an element will not be requested to provide more than BW/RTT chunks (where BW is the bandwidth from the element to the client and RTT is the round trip time from the element to the Client and back)**
5. **the system of claim 4, where if exists a segment of the file which has a set of known network elements that have this segment cached, and they've all provided BW/RTT chunks to the Client, they will provide more than BW/RTT chunks, pro-rata to their relative BW/RTT**

Typically, multiple Peers will be mapped to segments of the requested, where some of these segments will have several Peers mapped to them. it is therefore beneficial to be able to allocate which Peers will provide which Segments to the Client, in a manner that attempts to optimize the fastest total file retrieval speed, or in other manners that optimize other network parameters such as minimum distances travelled, etc.

Figure 76 suggests a method for optimizing the total download speed of a file, by choosing the faster Peers to provide more data than the slower Peers.

In 7602, a mapping of Peers to the file is received (either from local Client cache, from Agents in the network, or in any other method that is described in this patent or in Hola's patent 12836059 or in any other method).

Now there is a list of segments within the file, where for each segment there are zero or more network elements that may be used as Peers for this segment (or if this is a Tunnel list, this is a list of network elements that may load the information directly from the data server if requested). For each such segment in the list, an 'element_list' is created, which lists all of the network elements that have this segment cached (as per the mapping), where the elements are sorted according to their BW/RTT ratio (where BW is the bandwidth from the element to the client and RTT is the round trip time from the element to the Client and back).

It is preferable to request each segment from the element with the highest BW/RTT ratio, and this is indeed a sensible implementation. Thus for each one of the element_lists (i.e. the list of elements for each segment) 7606 starts with the first element in the list (i.e. the one with the highest BW/RTT ratio), and 7608 attempts to load the segment from that source. Thus it is can be expected that all segments will be loaded from the 'fastest' network elements. However, in practice, these network elements may be busy, offline, or otherwise unable to provide the information, in which case in block 7610 it is checked if additional chunks are needed to complete the segment request, and if so in 7612 the element number is increased, and it is

attempted in 7608 to then load the segment from the next element in the prioritized list 'element_list'. Once all segments are loaded (i.e. for each list in 7605, it reached the 'done' block in 7614), the file is fully loaded.

It is possible that the same network element will be the fastest network element to load all of the file from, and thus in case of a large file, the bottleneck of loading the file will be that element, whereas if several elements were providing the data for different segments of the file in parallel, that would make the file load faster.

It is therefore beneficial to have an algorithm for loading the file from multiple network elements in parallel. The 'optional' section of 7608, suggests that this may be accomplished by loading BW/RTT chunks of data from each element in the element list.

When the transaction is completed (i.e. the Client receives the whole file from the network and the Peers), the Client updates the Agent with the information on which Peer provided which segment. The Agent then updates the Peer map for this file in its memory, to provide in the future as the updated Peer map.

Peer Selection by Agent

When a Client queries an agent for a list of Peers for a specific file, the Agent searches within its memory for Peers that are listed as having parts or all of that file. This information may be collected by multiple means, for example by having Clients report to the Agent after they've loaded data, listing for the agent which Peers provided portions of that data, or by the Agent keeping track of which Clients have asked for information about a file (thus assuming that the Client has that file in its cache), or by getting the information about the list of Peers for files from other elements of the network.

CLAIM:

1. **in a system where Agents keep track of which Peers in the network may be storing data that is of use to other Clients in the network, the Agent keeps in memory which Clients have requested information about a file, in order to provide that agent as a Peer for future requests for information about that file**
2. **in a system where Agents keep track of which Peers in the network may be storing data that is of use to other Clients in the network, after the Client completes loading a file, it reports to the Agent that assisted it to learn about the Peers available for this file about which Peers actually provided which segments of the file, and the Agent then updates its information about the file and the Peers that have portions of it**

3. **the system in claim 2, where at the end of the transaction the Client updates all Agents that it currently has a connection to**
4. **the system in claim 2, where at the end of the transaction the Client updates zero or more network elements that it knows about with regards to the new map**

Typically, an Agent will have a large number of Peers for every file that it keeps track of. Transferring all of these Peers to Clients that request a Peer list from the Agent would result in lots of bytes of communications, thus congesting the web. Also, the receiving Client would most likely not use all of the Peers that the Agent is providing to it, and thus much of this communication is redundant.

It is therefore beneficial to have an algorithm to create a shorter list from the full list of Peers, that would best satisfy the Client's request. This algorithm should have two goals; (i) to determine how many Peers to provide to the Client, and (ii) to determine which Peers from the list to provide.

Figure 78 shows that in 7802 an Agent creates the list of Peers that have the data or portions of it that is requested by the Client. For example, a Client may be requesting information about a certain URL, and the Agent will respond with a list of Peers that contain portions of, or all of the data that is represented by this URL. This is done according to methods that are described in Hola's patent application number #12836059, and in other methods. The agent then prunes the peer list to X peers in 7804 according to various possible methods (some of which are hereby described), and then sends the pruned list to the Client in 7806. The Client then uses the peer list to fetch the data about the requested file in 7808.

(Note that in this patent, I refer to 'file' when I mean a piece of data which is requested by Clients from Data servers, and is then accelerated through the acceleration system. An example for such a 'file' is a URL – i.e. the data that resides in www.wiki.org/flower.jpg is a 'file' for this patent's purpose).

There are multiple ways that the Agent may determine the which Peers to send to the Client in the peer_list:

7812 shows a simple algorithm, of choosing the 'X' Peers that are closest to the Client (by geoIP, by IP, or otherwise).

7814 shows that these Peers may be chosen by those with the fastest connection to the Client. This could be implemented by choosing the ones with the highest estimated BW/RTT, where the estimation is done based on real measurements that the network elements do and report to the Agent, or by estimations based on any of known heuristics, such as IP distance.

7816 shows that it is possible also to choose the Peers by choosing those Peers that have most recently provided information about the file, since this may indicate a higher likelihood that the information is still cached on those Peers.

CLAIMS:

1. **in a system in which Agents keep track of which Peers in the network that may be storing data that is of use to other Clients in the network, when the agent needs to provide a list of peers to a requesting Client, and the agent has in its memory a longer list than will eventually be used by the Client, the Agent may reduce that list in size by choosing only the several Peers which are closest to the Client**
2. **the system in claim 1, where 5 Peers are chosen**
3. **in a system in which Agents keep track of which Peers in the network that may be storing data that is of use to other Clients in the network, when the agent needs to provide a list of peers to a requesting Client, and the agent has in its memory a longer list than will eventually be used by the Client, the Agent may reduce that list in size by choosing only the several Peers which have the highest BW/RTT ratio between them and the requesting Client**
4. **the system in claim 3, where the BW/RTT ratio is derived by feedback from other network elements that have measured this ratio in their communication with other elements**
5. **in a system in which Agents keep track of which Peers in the network that may be storing data that is of use to other Clients in the network, when the agent needs to provide a list of peers to a requesting Client, and the agent has in its memory a longer list than will eventually be used by the Client, the Agent may reduce that list in size by choosing only the Peers that have most recently successfully provided this information to other Peers**
6. **any combination of these previous 5 claims**

It is also beneficial to decide how many Peers should be provided in the list of Peers by the Agent to the requesting Client.

To determine how many peers are provided to the Client, a multitude of possibilities exist. For example, 5 Peers can be used (7820), or other numbers. Another alternative is to allocate a Peer for every X bytes, where a reasonable number would be 100kB for example (7822).

A third possible alternative of how to assign the Peers to the Client is described in the method in Fig 80. In this method, the Agent first creates a list of Peers (8002) that have the information that is requested by the Client, and places them in to a Peer list (peer_list). Next, the Agent sorts the list (8004) according to a metric of speed of delivery of the data between the peer and

the client. For example, this can be done according to the BW/RTT from the Peer to the Client. Next, the agent decides on a chunk size (8006) which will be the size of the pieces in to which the data is broken down to, for the sake of assigning pieces to the Peers to deliver to the Client (chunk_size). The chunk_size can either be a set value (for example, 16KB), or a number that changes according to the size of the data that is requested (for example $1/10^{th}$ of the size of the data itself). The Agent then assigns the chunks of data to each Peer in the following manner: starting with the first Peer in the list (8010), which is the fastest Peer because of the sorting in 8004, it then assigns a number of chunks to the Peer (8012) by using a metric of allocation. For example, by providing a set number of chunks per Peer, or by assigning for example BW/RTT chunks to the Peer. If there are more chunks of the data left to assign (8014), then the next Peer in the list is selected (8016), and more chunks are assigned to the Peer in 8012, and so on until all chunks are assigned, and the allocation function ends (8018).

CLAIM:

1. **in a system where Peers provide chunks of a file to a Client, a method for allocating the peers that will provide the chunks to the Client, and the number of chunks that each of these peers will provide to the Client, by providing each Peer a number of chunks relative to the speed of data delivery between the Peer and the Client**
2. **the method in 1, where the system of allocating the chunks to the Peers is by the BW/RTT between the Peer to the Client**
3. **the method in 1, where the system of allocating the chunks to the Peers is by allocating a fixed number of chunks to the Client**

Agent Allocation Algorithms (Passive Agents)

Before the acceleration system described in hola patent application number #12836059 can bring information from Peers, it needs to first find out who are the Agents that will serve this request. The stage of finding out from the acceleration server 1 who the Agents for this request are, consumes time, and therefore it is beneficial to reduce this time to a minimum, or eliminate this stage completely.

Figure 90 describes a method to reduce the time required for this step. First, the Client's status is checked 9002. If the Client requires Agents for a data request, then the Client checks 9004 in its cache to find out if it has cached one or more Agents (i.e. agent identifiers such as IP or host name) were used. At this step, it is also possible for the Client to send a request in parallel to the Acceleration Server 1, so that in case there are not enough Agents cached, a list is received from the Acceleration server as well.

If one or more agents exist in the Client's cache, these Agents are used 9010 as the Agent list that the Client queries for this request. If the Client also received Agents from the Acceleration server, it can add them to the list and query them as well for answers about the data requested.

At the end of the transaction 9012, the Client caches all the Agents that were used in this transaction, for future use. It may be beneficial to store additional information about the Agent, such as the connection speeds, the quality of the results, the resolved DNS, and other information that may help to connect faster to it next time, or to evaluate the connection next time, or to evaluate whether to try to connect to it next time.

It is also possible for the Client to occasionally receive a list of Agents from other network elements. For example, this may be implemented so that when network elements communicate with each other, they also exchange a list of cached Agents, or portions of them (for example, the last 100 Agents which they've communicated with are exchanged).

During the course of the operation of the acceleration system, a Client communicate with Peers that contain portions of the data that the Client needs to retrieve. It is typical that a Peer that has such portions of data that the Client needs to fetch, will have other portions of data that the Client will fetch in the future, since if two hosts requested a file (file A), and the first host also requested file B, there is a reasonable chance that the second host will later request file B as well. For this reason, if a group of Peers provided a host data, there is a chance that they also contain data that will later be requested by the host.

CLAIMS:

1. **in a system where Clients require to have a list of agents, a method by which the Client caches the Agents that are known to it and uses them for future requests**
2. **the method of claim 1, where the collection of agents is done by caching the agents that are received from the acceleration server**
3. **the method of claim 1, where the collection of agents is done by caching the agents that are received from other network elements**
4. **the method of claim 1, where the Client also queries the acceleration server for more Agents while working with its internal Cache of Agents**

Figure 92 offers a way by which the list of Agents may be received by the Peers that the Client already has communicated with, and also a way by which the Peers themselves may act as Agents for future requests by the host if they have the information that is required to be an agent.

When the Client is sending a request to the acceleration server to receive an Agent list, it looks up in its cache 9204 to find a list of Peers that it has communicated with, and creates a list [peer_list]. Optionally, this list can **be reduced to contain only the** Peers that have provided information to this Client in the past from the same data server as the current request is directed to. If (9206) there aren't any Peers in the peer_list, then the Client uses 9208 only the Agents received from the acceleration server. However, if there are Peers in the list, then the Client uses 9210 these as Agents, in addition to the Agents that it may get from the acceleration server. The Client may query all Agents in parallel, and use the fastest one or the one with the best information about Peers as the dominant Agent.

At the end of the transaction, the Client caches 9212 the Peers that were used in the transaction for future use, along with any other information that may be of help in the future, such as the data server, query, additional query information, speed information, etc.

CLAIMS:

1. **in a system such as hola's patent application number #12836059 , the Client may send out the request for agents also to a zero or more other network elements that the Client has communicated with in the past so that If these elements have knowledge of zero or more agents that may provide the information about the peers they will provide that agent list back to the client**
2. **in a system such as hola's patent application number #12836059 , the Client may send out the request for agents also to zero or more other network elements that the Client has communicated with in the past so that If these elements themselves have knowledge of zero or more Peers (including themselves) that may provide portions of the data that the Client needs they will provide themselves as an agent to the client**
3. **in a system such as hola's patent application number #12836059 , a method for acquiring an Agent by a Client for fulfilling a query intended for a data server, by which the Client may use a Peer that it has knowledge about as the Agent**
4. **the method described in claim 3, where the Peer is used if the Peer provided information about queries to that data server in the past**
5. **the method described in claim 3, where the acceleration server is queried for Agents in parallel**

Inline data

In an acceleration system such as described in Hola patent application number #12836059 , the overhead of getting the requested data from other Peers in the network is obviously higher than communicating directly with the data server. However, there are other benefits to the acceleration system which ultimately make it worthwhile in many cases. It is therefore

beneficial to reduce the overhead of the acceleration system as much as possible, to increase the benefit that it provides.

When a Client 2 in an acceleration system requests a small file (for example a file under 16kB) which exists in a networked element such as an Agent or Peer, it is beneficial that when the Client asks the Agent or the Peer for information about this data, that the Agent or Peer simply communicate the data itself immediately, instead of providing the meta data (e.g. where it is located, or which part of it the Peer has).

Figure 94 is a flowchart of a method of providing small file data 'in line' to the communication stream. First, the Client's action is evaluated 9402. If the Client is sending 9404 a request for URL to Agent, then that request for meta data is sent (i.e this is a request about which Peers may have this data). Once the reply from the Agent is received back at the Client, the software checks 9406 whether Meta data or inline data was received. If Meta data was received, then the software proceeds 9208 with the normal course of operation. If the Client receives Data from the Agent (which then means that the data is probably small), then the Client uses the data 9210 as if it was received from the Peers, and there's no need for further action on this data.

If in 9402 it is detected that the Client is receiving data from a Tunnel, then the Client checks 9412 whether the data is smaller than the set size for what a small file is defined as (for example 16kB). If the file is not smaller than that size, then normal operation proceeds 9214. If however the data is indeed below that certain size, then the Client also sends 9216 this information to the Agent or group of Agents that are helping the Client, so that they can send this information later to other Clients that request this data.

CLAIM:

1. **in a system such as that in Hola's patent application number #12836059 , a method by which when the Client queries the Agent about a file, the Agent may provide that file directly to the Client instead of providing the Client with the meta data about the file**
2. **the method in claim 1, where the data is only provided when the file is below a certain size**
3. **the method in claim 2, where that size is 16KB**
4. **the method in claim 1 where if a Client loads data from any other source and that data is below that certain size, it updates the Agent with the data itself and the Agent stores that data for providing it to a Client in the future**

Providing the information in line requires implementation within the software that operates the Agent as well. Figure 100 is an example of such an implementation.

In 10002 the Agent receives a message from a Client and checks which type of message it received. If it received a request for a URL, it checks 10004 its cache for whether it has information about this data request. If it does have an entry in the cache for this data request, it checks whether this entry stores the Meta data about the request, or the data itself that answers the request. The Agent sends back to the Client the Meta Data or the Data accordingly. If it is sending back the Data, it marks it as Data for the Client to process accordingly.

If however the Agent received 10006 Data from a Client, then this data is to be stored within the Agent's cache together with the data request that is provided as well from the Client, to be provided in the future to Clients requesting information about this Data.

If the Agent receives 10008 Meta data from a Client, then it proceeds normally to store this meta data along with the data request that is provided as well from the Client, to be provided in the future to Clients requesting information about this Data.

Tunnel caching

In figure 102, Client_1 10202 and Client_2 10204 are both Clients that are accessing data from the data server 5 through a Tunnel 10206.

It would be beneficial for the Tunnel 10206 to cache the data server 5 responses to the data requests that it receives from the Clients, so that if the Tunnel receives such queries in the future.

Figure 104 is a flowchart showing how the Tunnel caches requests and provides the results to other Clients that make similar requests.

When the Tunnel receives 10402 a request for data from a Client it first checks 10404 whether this Tunnel already has received such a request in the past or if its currently serving such a request, in which case it may have some of the data cached, and thus may provide it from the cache instead of fetching it from the Data Server. If indeed the request has been made then in 10410 it checks whether the response for this request is cached within the Tunnel, and if it is, then it is sent 10412 back to the Client. The Client will then update the Agents with which it is working so that they will update their databases to include the responding Tunnel as another potential Peer for this data response for future such requests. In 10406 the request is requested from the Data Server, and the result is cached in the Tunnel, and then in 10412 sent back to the Client.

(tt 1.5h)

CLAIM:

1. **in a system such as that in Hola's patent application number #12836059 , a method by which when the Client sends the request to the Tunnel, the Tunnel may provide back results from its cache**

Using Back Ends to Customize Specific Data Requests

Figure 120 shows an system where Applications 8 are sending requests to a Module 10, the module 10 deals with the requests coming to it from the Applications 8 in a standard manner. However, there is a Back End Module 11, which defines certain criteria under which the Module 10 will handle the requests in a different manner. The manner of handling these "non standard" requests may be defined within the Back End Module 11 or within the Acceleration Module 10. The Back End Module 11 has a Back End Database 12004 which includes a list of criteria for requests which special handling, and a list of functions that describe the handling of the requests that meet these criteria.

In the preferred embodiment, where the Acceleration Module 10 is an acceleration system such as that described in Hola's patent application number #12836059 , the back end module would define criteria for data requests where it would be more beneficial to treat these requests differently than the other requests that flow through the Acceleration Module. For example, if the Application 8 is a Voice Over IP application (such as Skype), it may be useful to open 2 different routes to the other side of the call, and to send every packet through both routes in parallel, in order to increase the reliability of the packet transport. This would require the Back End Module 11 to define a criteria (in this case – for all VOIP calls), and a behavior (in this case, to open two parallel connections and to transmit each packet through both routes).

Block 12002 defines various ideas for criteria for different handling by the Back End Module 11. This list is an example of ideas, but does not list all of the ideas for different criteria.

Here are some examples of who may benefit from different treatment as described:

1. by URL: specific files on the Internet (e.g. "Wikipedia.org/contact.html")
2. by Domain: specific web sites (e.g. "Wikipedia.org")
3. by Data server IP: specific servers (e.g. "208.80.152.201")
4. by Type of file: specific file types (e.g. ".flv files")

5. by time of day: specific handling of all files or a group of files during certain hours of the day (e.g. "all files between 11pm to 4am")
6. by Geography of Client: specific handling according to where the Client is (e.g. "for all Clients in Germany")

CLAIM:

1. **in a system such as that in Hola's patent application number #12836059 , a method by which when the Client sends the data request to the Acceleration Module, a Back End module will apply certain criteria to the requests, and requests that pass the criteria will be handled by a method that is the handling method for that criteria**
2. **the system in claim 1, where the criteria is that the data request is all requests for certain URLs**
3. **the system in claim 1, where the criteria is that the data request is all requests for certain domains**
4. **the system in claim 1, where the criteria is that the data request is all requests for certain data server IPs**
5. **the system in claim 1, where the criteria is that the data request is all requests for certain types of files**
6. **the system in claim 1, where the criteria is that the data request is all requests made during a certain time range during the day**
7. **the system in claim 1, where the criteria is that the data request is all requests made by clients from a certain geography**

Figure 122 is a flowchart describing the operation of this system. In 12202, every request from the Application 8 to the Acceleration Module 10 is evaluated separately - each of the criteria in the Back End Database 12004 is applied to the request to test for a match. For example, one of the criteria may be to check if the request is going out to Youtube.com, so that different handling of YouTube videos may be applied.

In 12206 the evaluation is done, and if there are not any criteria in the Back End Database for which the data request from the Application 8 is a fit, then no action is taken 12208. If however, there is a fit for the data request with one of the criteria, then the function that is associated with this criteria is applied to the data request in 12210.

On the Internet, web sites are interested in learning about their usage statistics (i.e. understanding how many page views their various web pages are receiving, average time between page changes, etc.). This is many times done by embedding a piece of code within their web pages, which causes the viewer's web browser to send a request to an analytics server, such as the Google Analytics Server, which then may provide them the statistics that

they seek. The downside of this is that the web browser needs to wait for a response from the analytics server, and this sometimes slows down the loading of the page. In such a case, it may be beneficial to employ a specific "back end" for analytics servers (also called "usage stats" sometimes).

Figure 124 is a flowchart for an example of such a specific back end implementation, for analytic servers. In 12402, a request to a specific analytic server is identified. This may be identified by the IP of the server, by the URL of the file that the browser is requesting from the analytics server, or by any other method.

Each different analytics server typically has a standard response to such a request. This standard response is stored in the stats_backend_database, which is a database that for each analytics server, it stores the request to each request that may be asked of that server or URL as the case may be. This can be achieved by either collecting this information centrally (either manually or otherwise) and loading this information in to the Clients' databases, or by looking up the request to the server in the database, and if it does not exist to ask the request directly from the server, and to cache its response for the next time this is requested.

In 12404, the request is compared to the requestes that exist in the stats_backend_database, and the typical response from this server is retrieved from the database, and returned to the requesting application in 12406 (typically to the web browser), such that to the requesting application it seems that this request was received from the analytics server, so that the browser can complete this transaction. In 12408 the original request is sent to the analytics server (among other reasons, this is done so that the analytics server still has the right information about the web site). **An improved implementation may choose to only send this request to the analytics server when the network connection for the Client device is idle, in order not to take up bandwidth required for more urgent uses.**

In 12410 the back end waits for the response from the analytics server, and discards it once it arrives. **An improved implementation caches this result within the database, so that next time the browser makes a similar request, this result is returned, thus the results in the database are up to date.**

CLAIM:

8. **the system in claim 1, where the criteria is that the data request is to an analytics server**
9. **the system in claim 8, where the module returns the typical answer from this analytics server to the requesting application, where the typical answer is stored in the module's cache**

10. **the system in claim 9, where the request is also sent to the analytics server, and the response from the server is not provided back to the requesting application**
11. **the system in claim 10, where the module's cache of typical answers is updated by the answers received by the analytics server**

Another embodiment of a specific back end is shown in figure 126. Typically, videos on the Internet, such as the case with Youtube.com videos is that the site will have its own method of identifying the video, and the offset of the viewing point within the video itself. For example, the video that is identified by the URL 12620 (HTTP://WWW.YOUTUBE.COM/WATCH?V=9mSb3P7cZlE?ST=1:48) includes several parts: 12602 is the Domain of the URL (www.youtube.com) – i.e. the name of the server from which to load the movie. 12604 is the identifier of the video within the domain, in this case the video's identifier is 9mSb3P7cZlE, and 12606 is the offset (in seconds) that the user is interested to watch – in this case the user is interested to watch the video starting at the 1 minute and 48 second spot.

This method, which may be different from one video site to another, presents several problems. One problem is that when a user requests a video with a time offset, a unique URL is generated (such as the one referenced by 12620), and caching systems cannot identify that this video has already been cached by their systems. Another problem is that offsets may be sometimes denoted in minutes and seconds (such as in 12620), sometimes in byte offsets, sometimes in relative offset (%), and in other methods. For caching systems to be able to cache this, it is important to have a method to normalize these parameters.

Thus, it is beneficial to develop a method that receives as an input the URL of a video, and can then identify which video file this is referencing, and can normalize the offsets so that existing cached copies of the file can be provided to answer the request.

CLAIM:

1. **a system that comprises of a software module, a database, where upon getting a url as an input, the system looks up the domain of the URL within the database, retrieves the schema of the URL and can identify the identity of the video, and the offset to view within the video according to the schema**
2. **the system of claim 1 where the file is cached according to the video identifier**

Figure 128 is a flowchart for how the process of creating a unique identifier for the URL can be implemented. In 12802 the domain 12602 is looked up within the video back end database. This database lists for each such domain, the schema for how that domain represents URLs for

Videos and the offset within the videos. This schema defines for the module how the URL is broken down in to its parts: The domain (DOMAIN), the video identifier (VIDEO_FILE_IDENTIFIER) and the offset within the video that the user is requesting to view (OFFSET).

12804 then creates a new identifier from this information, consisting of the DOMAIN, appended with a "/", then appended with the VIDEO_FILE_IDENTIFIER. Thus, for example, the video referenced by the URL: HTTP://WWW.YOUTUBE.COM/WATCH?V=9MSB3P7CZlE?ST=1:48, becomes WWW.YOUTUBE.COM/9MSB3P7CZlE.

If this is implemented within a caching system, then the video can then be cached along with its new identifier, and the start and end OFFSET, to be used to serve to other requesters of ranges within this content. For example, if the range cached is from 1min to 22min, and a request is received for the range 15min to 17min, this request can be answered by this cache, since this range is within the cached range.

There are several options for an offset within a URL for a video. Three such popularly used offsets are illustrated in Fib. 130: 13002 is a time offset, that can be denoted as #t=3M54S for example, which would state that the offset is at 3 minutes and 54 seconds. 13004 is a Byte offset, that can be denoted as B=10344 for example, which would mean that the offset within the video file is 10,344 bytes from the start of the file. 13006 is a relative offset, that can be denoted as R=54.3 for example, which would mean that the offset is at the point which is 54.3% of the video (for example at byte 543 of a 1000 byte video).

If a URL is received with one of these offsets is received, it is typically not possible to be able to normalize (convert) the URL to be with another form of offset, because some parameters are missing. For example, if a relative offset is received (e.g. R=54.3), it is impossible to convert this in to a byte offset without knowing how many bytes the video is comprised of.

Figure 130 is a flowchart for how to normalize the OFFSET within the URL of the video. In 13008, the URL with the offset is received (could be for example received by a caching system by a client that is requesting to load the file from the cache if it exists). In 13010, the URL is normalized in to a URL according to the method in Figure 128. In order to get the additional information required in order to normalize this URL, 13012 sends a request with this new URL to the data server, and closes the connection after receiving back the initial part of the response from the data server that includes the full file meta-information, including the content length in bytes, and in minutes/seconds, to be able to normalize the offset. In 13014 the new offset is calculated. This can be done for example by always converting the OFFSET received within the URL of the video file to a relative offset (for example if received a time offset, dividing it by the total time of the file, or if it was a byte offset, dividing it by the size of the file

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 22 of 85

in bytes, or if it was a relative offset, leaving it as is), and then multiplying that number by the size (in bytes) of the video file, and then making the request in a byte offset to the cache or other cache or data systems. In 13018 the new URL with the new byte offset is used to retrieve this video file in caching systems in a normalized form, or to retrieve the same.

1. **a system of calculating the relative offset of a given video file with a given offset in non-relative units, by requesting the video file from the data server without specifying an offset, getting the full file information from the response, and dividing the non-relative offset by the file's real size**
2. **the system of claim 1, where the connection to the data server is closed after receiving the file information**

Increasing number of connections to data server via tunnels

Data servers typically limit the number of active connections that a host can keep open with them concurrently. This limitation is detrimental in various scenarios. One such scenario as an example is where a communications device with a web browser needs to load many URLs from a Data Server in order to display a web page and thus would optimally like to open more connections to the Data Server to load these URLs than the server allows one host to open. In such cases, a system that would allow more than the allowed concurrent connections to exist would be beneficial.

Figure 136 describes such a system. In it, the Client 2 would like to open more connections with the Data Server 5 than the Data Server 5 will allow (where each of these connections is loading one or more requests, where each request is a resource, a range of a resource, or multiple resources). The Client then communicates with participating Tunnels 4 (i.e. network devices on which software is installed for this purpose), and requesting from each of these Tunnels to load several of these requests, such that the each of the participating elements (the Client and the Tunnels from which it requested the loads) does not open more connections vs. the Data Server 5 that the Data Server 5 allows.

For example, if the Client 2 would optimally need to open 15 connections to the Data Server 5, but the Data Server limits the number of open connections from a Client to 8, then the Client 2 opens 8 connections to the Data Server 5, and uses a Tunnel 4 to open up the additional 7 connections to the Data Server 5.

Claims:

1. **a system that comprises of a Client communication device that is loading data from a Data Server and one or more Tunnel communication devices, where when the number of connections that the Client would benefit from opening vs. the Data Server is higher than the number of allowable connections, the Client requests for some of the connections to be opened by one or more Tunnels to the Data Server and for the Tunnel to return the retrieved data back to the Client device, so that each of the participating Tunnels and the Client are not opening more than the allowable connections to the Data Server, but opening up the desired amount of connections in aggregate**

Figure 138 is a flowchart showing a possible implementation of this. When a Client device is communicating with a Data Server, the Client determines 13802 the maximum number of connections between the Client and the Data Server that it will use (MAX_CONN). This may be done for example by deciding on a number that will always be the MAX_CONN. In HTTP for example this number can be 8 connections. Other ways to determine this number is by periodically updating the Clients' MAX_CONN to be the number that is currently used by software on Data Servers, or specifications that may arise from time to time.

While the Client makes a new request to the Data Server, it evaluates 13804 whether the new request will be done on an open connection to the Data Server, or whether opening more connections to the Data Server will be beneficial for this new request. A new connection may be beneficial for example if the current connections are busy with prior requests, or if opening an additional connection may be beneficial for more bandwidth, more quality of server, overcome server overload, etc. A trivial implementation of this may be for example to open a new connection every time a current connection is involved in a pending transaction (i.e. waiting to transmit or receive information from the Data Server).

If it is not deemed beneficial to open a new connection to the Data Server, then the data is transmitted 13820 over one of the current open connections. If it is deemed beneficial to open a new connection to the Data Server, then the Client checks 13808 whether the number of connections between this Client and the Data Server has reached the maximum allowed connections. If not, then a new connection is opened 13806, and the data is transmitted to the data server 13820.

If the number of open connections to the Data Server is at the maximum allowed 13808, then the Client chooses 13812 a Tunnel from the list of Tunnels that the Acceleration Server has provided to the Client. If such a Tunnel does not exist, or previous attempts to get the information from this Tunnel failed, then the Client requests another Tunnel from the Acceleration Server.

The Client then sends 13814 the request that was intended for the Data Server to the Tunnel that was chosen. The Tunnel will then either send the request to the Data Server, and provide the response from the Data Server back to the Client, or the Tunnel will reject the request (for example if the maximum allowed connections to the Data Server are already open).

The implementation on the Tunnel side is simple; when receiving a request from a Client, forward the request to the Data Server, and provide the response back to the Client. This is implemented as a regular proxy, of which many open source implementations exist in the market. The only difference to be implemented in the Tunnel is that if the Tunnel has the maximum allowed connections to the Data Server, the Tunnel rejects the request to the Client, and the Client will then use a different Tunnel. In 13816 the Client checks the response of the Tunnel, and if the Tunnel rejected the request, then another Tunnel is chosen in 13812.

If the Tunnel did not reject the request, then the Client waits 13818 for the response to come back from the Tunnel after the Tunnel received it back from the Data Server.

Agent selection; feedback to server about Agents

Figure 142 shows a system such as Hola's patent application number #12836059 , in which when a Client 2 needs information from a Data Server 5, the Client 2 requests and receives a list of Agents 3 from the Acceleration Server 5 for this request. The Agent list is received typically by receiving a list of host names. Having more information about this server list would be beneficial for the Client. For example, if the information about each of the Agents included the Agent's IP address, then the DNS lookup that the Client typically needs to make on this host name would be saved. If for example the information includes information about the speed at which each phase of the connection with the Agent takes (TCP connect, etc.), then the Client can better estimate the 'cost' of connecting to each Agent in terms of time.

Figure 146 is a flowchart for an implementation of the feedback cycle. The Client sends 14602 to the Acceleration Server information about the request it needs to make to the Data Server, and receives back a list of Agents to use for this request 14604, and sends this request to each of the Agents.

The Client completes the request as per Hola's patent application number #12836059 . Once the request is completed, the Client sends 14608 back data about the Agents and their performance to the Acceleration server for future use. This information may include the IP address, ports used, speed for each of the connection phases, and whether the agent had the information that the Client needed. Optionally, the Client can wait to send this data only when

the network is free from communications, so that this information does not lower the performance of the more time critical data that may be flowing on the network.

On the server side, the implementation is such that the server caches the information about each of the Agents, and provides this information to the Clients when it provides them with an Agent list. In addition, this information may be used by the Server to decide which Agents to send to the Clients, for example, the fastest performing agents, or the ones that provided the highest percentage of answers to Clients in the past, etc.

Figure 144 is a flowchart of an implementation of the server side implementation of the Agent selection feedback. The acceleration server receives 14402 the request from the client to provide agents. The acceleration server chooses 14402 the X best Agents (X can be 5 for example) near the client. For example, this can be done by evaluating the 200 Agents which are the closest to the Client in terms of their IP, and from them choosing the X best Agents for this request. Also in 14404, the AC chooses an additional X Agents from within this range which are 'second best' and provides them to the Client as well. The AC then provides 14406 the Agent with all of the information that it has cached about these Agents. The AC checks 14406 for a response from the Client with information about the Agents from this current transaction. The Client response 14408 will include all connection information for all these agents (which are online, offline, response for each stage in the connection, information about data received, IP address, etc.). The AC caches 14410 this information for use for future recommendations to Clients, and for providing more information to them in the future.

Optionally, the Clients may provide to the AC, and the AC may cache, information about the Peers used, (IP address, connection times, etc.), that the AC can cache for when these Peers are to be used as Agents for other requests.

CLAIM:

1. **in a system such as hola's patent application number #12836059 , the Client receives a list of Agents to use, uses them for a specific transaction, and when the network is idle, sends back to the Acceleration Server information about these agents for the server to use in recommending these agents in the future**
2. **the system in claim 1, where the information is sent only when the network is idle**
3. **the system in claim 1, where the information provided includes one or more of the agent's IP, ports used, speed for each of the connection phases, and whether the agent had the information that the Client needed**
4. **the system in claim 1, where the acceleration server uses this information by providing back this information to the client to use in transactions with the agent**

5. **the system in claim 1, where the acceleration server uses this information to decide which agent to recommend in future requests by clients**
6. **the system in claim 1, where the Clients may also provide back information to the Acceleration Server about Peers used in the transaction, for the Acceleration Server to use when providing these Peers as Agents to requesting Clients**

Feedback about peers from client to agent and between agents on information they cache

Figure 148 shows a P2P system such as Hola's patent application number #12836059 , by which a Client 2 sends a request to a few Agents (in this example to 14802, 14804 and 14808), and ultimately chooses to work with one Selected Agent 14802, the Selected Agent 14802 is typically chosen because the Client determined that it would be able to get the information required from the Selected Agent 14802 faster than from the other Agents. This could be due to the speed of communications between the Client and this Agent vs. with the other Agent, or it could be because the Selected Agent has information that would be more useful than what the other Agents have. In such a case, it would be beneficial that after the transaction, the information used from the Selected Agent 14802 would be provided to the other Agents (in this case to 14804 and 14806) for them to cache for future transactions.

In a P2P system such as Hola's patent application number #12836059 where the required information is the list of Peers that can satisfy the Client's data request, it would be beneficial for the Client to use those Peers and to then update the Agents about the Peers, adding to the Agents additional information that it collected during the transaction with the Peers.

CLAIM:

1. **in a P2P system, a comprised of a Client that requests information from a group of one or more Agents that may possess the required information, a method by which after Client receives this requested information from one or more of the Agents, the Agents in the group that did not have the requested information are provided that information to cache for future user**
2. **the system in claim 1, where the P2P system is Hola's patent application number #12836059**
3. **the system in claim 1, where the Agents are updated by the Client**
4. **the system in claim 3, where the Client updates the Agents after the end of the transaction, so that if there is more updated information to provide to the Agents based on this transaction, it is provided to all the Agents in the group**
5. **the system in claim 3, where the updated information is only provided when the Client's network connection is not busy with other data**

6. **In a P2P system such as Hola's patent application number #12836059 , a method by which the Client uses the group of one or more Peers, and then updates the Agent that provided the group to the Client about the additional information learned about the Peers**
7. **The method in claim 6, where the information learned is one or more of whether the Peer had the data that the Client needed, the connection time to the Peer for each phase of the connection and the IP of the Peer.**

Figure 152 is a flowchart for the client side implementation of how a Client updates the Agent from which it received data about Peers, with new data learned during the transaction. The Client sends 15202 a request to the Agent to receive a Peer list for this request. It then 15204 uses those Peers for its transaction as per Hola's patent application number #12836059 for example. During the transaction, the client logs 15206 the information learned during the transaction about each Peer that it uses, and compares that information to the information about the Peer that it received from the Agent. The Client makes a list of differences between the information it received from the Agent relative to the information it learned during the transaction, to use later to update the Agent. The elements in the P2P network would all need to be configured to log and update specific information about the transaction. This can be for example only whether the Peers are online or not, or can include much more detailed information, such as the IP of the Peers, the Ports they are using, the information that the Peers have, and more.

At the end of the transaction, the Client transmits 15208 the list of differences that it created back to the Agent so that the Agent can provide more accurate information to other network elements in the future. It is possible to wait with this update until the Client's network connection is not busy, so that more important data will not be delayed.

Figure 154 is a flowchart of the Agent side for the updating of the Peer information. The agent 15402 receives a request from a Client to provide Peers for a data request. The Agent provides 15404 the Peer with the X most preferable Peers (as per Hola's patent application number #12836059), and provides the Client with other information about these Peers that can help the Client during the transaction. This can be at the minimum the hostname of the Peer, or for more advanced implementations may include the IP of the Peer, when it was last online, and more.

The Agent then waits 15405to receive feedback from the Client about the Peers. Once received a response from the Client, the Client response will include 15406 updated connection information about the Peers. The Agent then caches 15408 this information to provide to other network elements that may request this information, and to provide to clients when providing them with a peer list (such as in 15404).

Load balancing

Load balancing is a computer networking methodology to distribute workload across multiple computers or a computer cluster, network links, central processing units, disk drives, or other resources, to achieve optimal resource utilization, maximize throughput, minimize response time, and avoid overload. Using multiple components with load balancing, instead of a single component, may increase reliability through redundancy. Load balancing is explained in the document at http://en.wikipedia.org/wiki/Load_balancing_(computing) and is incorporated in its entirety for all purposes as if fully set forth herein.

It is beneficial to create a method by which it is possible to efficiently monitor a group of resources assigned to some task, and to be able to add or reduce resources from the group according to the congestion of that group. When the resources are congested they signal a central server, and if over a certain amount of the resources signal that they are congested, adds a resource to the group, and if over a certain amount of the resources have not signaled congestion the server removes one or more resources from the group of resources.

Figure 156 shows an acceleration network such as that in Hola's patent application number #12836059 , which is one example of where load balancing may be used. In Figure 156 a group of Agents 156004 is allocated to a Data Server 5. The Agents 5 within the group 156004 may be subject to an increase in load from Clients (such as 2 in this diagram), and thus to need to balance the load between them, or to have more Agents added to the group 156004 in order to handle this additional load. It is also possible that the load of the Agent Group 156004 goes down, and the group has free resources. In such a case it would be beneficial to reduce the amount of resources of the Agent Group 156004 so that these freed resources can be free to server other Data Serves.

In Figure 158 an example implementation of such a load balancing system is provided, within an acceleration system such as Hola's patent application number #12836059 . In 15802 the server checks if it has received a message. If it hasn't received a message it continues to wait for one. If a "congestion" message has been received from an Agent, then in 15804 this Agent is mared as 'congested' in the acceleration server's database. Then it is checked 15806 whether over 20% of the agents allocated for any of the servers that this agent is serving are marked as congested. If they are not, then nothing is done and the server continues to wait for a message in 15802. However, if over 20% of the agents that are assigned to a particular data server are indeed marked as congested, then the Acceleration Server allocates 20% more Agents for that

particular Data Server by selecting more Agents from the available nodes (in any algorithm that is chosen – this does not matter), and marking them in the database as assigned to this particular Data Server. Note that 20% is an example implementation. Other numbers will work as well. Then the server continues waiting for the next message.

In 15808 the Acceleration Server receives a request from a Client that seeks to communicate with a Data Server, and the Client requests to receive a list of Agents through which to communicate. The Acceleration server then creates 15810 a list of Agents that are assigned to this particular Data Server according to its database. It then checks 15814 the number of Agents in this list. If there are under 5 (example implementation – can be other numbers as well) Agents assigned to this Data Server, then the Acceleration server allocates 15816 more Agents to this Data Server to complete the list to 5 Agents, and sends 15818 the list to the client, and resumes waiting for additional messages. If the list created in 15814 is of more than 5 Agents, then the Acceleration Server checks 15820 how many of these Agents in the list that are assigned to this Data Server have transmitted a congestion signal to the Acceleration Server in the past 10 seconds (10 is an example, could be other times as well). If 80% (for example) of the Agents in the list have not transmitted a congestion message within this timeframe, then the Acceleration Server removes 15822 an Agent from this list, and marks it within the database as not assigned to this particular Data Server any longer, and then sends 15818 the list of Agents to the Client. If in 15814 there are exactly 5 Agents in the list then these are sent 15818 to the Client.

CLAIM:

1. **A method for managing congestion within a group of elements, where if a central load balancing server identifies that an element is congested then if over a certain amount of the elements within the group are congested, it adds an element to the group, and if over a certain amount of the elements have not signaled congestion the server removes one or more elements from the group**
2. **The method in claim 1, where the signaling to the central load balancing server is done by the congested elements**
3. **The method in claim 1, where the signaling to the central load balancing server is done by the communication devices that are trying to use these elements**
4. **The method in claim 1, where the signaling to the central load balancing server is done by the elements whenever their own resources are used over a certain threshold**
5. **The method in claim 4, where the resources are one or more of its storage, I/O, CPU, bandwidth**

6. **The method in claim 1, where the central load balancing server sends a request to the elements and marks them as congested if they do not respond within a determined timeframe**

Figure 160 provides more methods that a given system may learn about congestion within its elements. 16002 is a method of learning congestion by the nodes within the system, whereas if nodes are querying servers and not receiving responses, then they can directly report to the central load balancing server that the particular resource is congested, thus triggering the central load balancing server (the Acceleration server in the case of Hola's acceleration system in patent application number #12836059) to allocate more resources to the congested resource group.

In 16004 the Agent checks its own resources (storage, I/O, CPU, Bandwidth, etc.) and if they are being used over a certain threshold, (e.g. above 10% of the CPU for example), then the agent sends a 'congested' signal to the acceleration server

In 16006 the acceleration server checks for congestion by pinging the Agents (or sending them some other type of request) and marking them as congested if they do not reply above a certain time (e.g. above 300ms)

In Fig. 162, Element 1 162040 is handling requests from multiple IPs – in this example from Device 1 162002 to Device 8 162016. In the event that Element 1 is congested (i.e. has more work load than is desired), the work load should be re-distributed between the congested element (in this case Element 1 162040) and another Element which should be allocated (in this case Element 2 162044. An algorithm is needed to find other Elements that are free to share the work load with the congested element, and to re-allocate the work load between the two elements. It is also required to define how the new responsibility is communicated to other elements of the system.

Fig. 164 is a flowchart describing the algorithm for load balancing between elements. In 16402 the element is defined as congested (either by internal means such as checking its CPU load, its network load, its disk access load, etc.). The Element then sorts 16404 all the devices that have accessed it according to their geo IP, so that x comes before y in the list, if geo_ip(x) < geo_ip(y). The Element then chooses 16406 the Device that is the 'middle' of this list. The middle of the list may be determined in a multitude of methods, such as the IP of the median Device in this list, or the IP in the middle of the range, or in a preferred embodiment, the IP for which above and below this IP the number of requests coming in to the Element are the same. In 16408 an element is chosen to assist the congested element. This new element is chosen from a list of candidates that can be assembled in a variety of ways. For example, this could be any one of the following: (i) any one of the free elements (i.e. of elements not currently assigned to any ip

range), or (ii) a free element that is close to the devices accessing it, or to the data server it is accessing, or (iii) an element that is not free, but is less loaded than this element where the metrics for load may be a variety of methods known to those who are skilled in the art, or (iv) by notifying a central server to allocate element to the other range and getting that element from the central server. The congested element then assigns 16410 all the IPs up to the middle IP (i.e. IP(1)…IP(MIDDLE) as IPs that it will handle, and assigns the rest of the IPs to the new element

If the congested element receives future requests for handling the range that it has assigned to the new element, it will not answer these requests, and instead will redirect them to the new element in any of a multitude of methods that are apparent to those skilled in the art.

Figure 166 provides several improvements in allocation of elements to tasks. In 16602, a central server (for example the acceleration server in hola's patent application number #12836059) periodically asseses which of the elements are under a certain threshold of congestion, and will merge the responsibilities of two or more such elements in order to build a bigger list of free elements that can later assist elements that are congested. For example, the acceleration patent in Hola's patent application number #12836059 can periodically review which of the agents has received under a certain number of requests in the past X seconds (e.g. 180 seconds), and merge the range of responsibility from that Agent with another such Agent who's load is also under that threshold.

In 16604, instead of logging all the ips that are accessing an element, the element can log only the x recent requests (x could be 1,000 for example), or keep a logarithmic random list of x from the past 24 hours, x/2 from past week, x/4 from past month, etc.) in order to reduce the amount of memory required to assess the range of IPs that are currently accessing this element.

Claim:

1. in a system such as hola's patent application number #12836059 , the acceleration server periodically reviews the load of each of the Agent elements, and if two elements are under a certain threshold of load, it merges the range responsibilities of both Agents
2. the method in claim 1, where the responsibilities are the Peers that the Agent is responsible for serving
3. the method in claim 1, where the responsibilities are the Data servers that the Agent is responsible for serving
4. a method by which in a system where a network element needs to log the elements that are accessing it, it only logs a portion of the requests according to a certain algorithm, by which the list of accessing elements will be representative of the overall requests

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 32 of 85

5. the method in claim #4, where the algorithm is by logging only the past serveral requests (several can be 1000 for example, or from the past 5 minutes)
6. the method in claim #4, where the algorithm is by logging a list of logarithmically reducing list of random requests

Micro chunks

Typically, in systems where a client retrieves chunks of a file from various sources (such as in Hola's patent application number #12836059), the client is optimized to retrieve a different number of chunks from each of the availalble sources, such that from sources that can provide information faster to the client, more chunks are retrieved. However, this optimization is dependent on the size of the chunk, and sometimes having a different sized chunk would make a much more efficient system. Figure 170 shows such a prior art system, in which 17001 is a file that the client 2 requests to load. This file is comprised of 6 equally sized chunks. In this example, the client 2 loads these chunks from 3 available sources, fetching more chunks from faster sources, so that 3 chunks are requested from source 1 17002, 1 chunk from source 2 17004 and 2 chunks from source 3 17006. If the client 2 completes loading the chunks from source 1 17002 and from source 3 17006 loading before the chunk from source 2 17004 completes the load, then source_2's chunk is delaying the loading of the file_to_load to the client 2. In such a case, it would be beneficial to split the chunk requested from source 2 in to smaller chunks, and request some of those chunks from other sources.

Figure 172 shows the prior art system from Figure 170 after the chunk sent to source 2 17004 was split in to two smaller chunks, one of them sent to source 2 and the other to source 1.

Figure 174 is a flowchart for deciding when and how to split chunks in to smaller chunks in such a system. In 17402 the client is going to load a file which is available bia multiple sources (source(1)…source(n_sources). In 17404, the requested file is split up in to N chunks of equal size (chunk_size) that was pre-determined in the system. For example, this chunk size can be 16KB. In 17406 each source is assigned a number of chunks according to the reverse proportion of the expected time it would take the client to load a chunk from that source (i.e. the faster the source relative to the other sources, the more chunks will be requested from that source relative to the other sources). In 17408, the program evaluates the time that the last chunk is expected to be received from a source, relative to the one-before-last chunk (i.e. if there are N chunks, the program evaluates the time at which N is received (T(N)) minus the time at which the previous chunk arrives (T(N-1)). The evaluation of the time to receive a packet from a source may be done in various ways, including ways of evaluating bandwidth (BW) and (RTT) from sources on the network that are mentioned in some of Hola's patents

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 33 of 85

(reference to all our patents here). In 17410, it is checked whether the difference between T(N) and T(N-1) is larger than E, where E is the threshold determined for invoking a splitting of the chunk size. E can be for example measured in time (e.g. 50ms), or in relative time of the complete download (e.g. 10% of the total time), or for example as half the time it would take the source to provide the chunk (0.5 * CHUNK_SIZE/BW + RTT). If the time is under that threshold, nothing needs to be done. If the time is over that threshold, then in 17414 the chunk size is reduced (in this case by half, but other implementations may lower the chunk size by other measures), and in 17416 one of those two new chunks is assigned to the source, and the other new chunk is assigned to the source who is expected to be able to deliver that chunk the fastest.

CLAIM:

1. **In a system where a client requests chunks of a file, each larger than some minimum size, from various sources, and where it is expected that one of the sources will finish providing its last chunk to the client much after the all other chunks have been loaded from the other sources, then this chunk is split in to smaller sized chunks, and re-distributed between the various sources**
2. **The method in claim 1, where the smaller sized chunks are half the size of the previous size**
3. **The method in claim 1 where the splitting up in to smaller chunks is only done if the last chunk is expected to delay the full load by 10% of the full load time**
4. **The method in claim 1 where the splitting up in to smaller chunks is only done if the last chunk is expected to delay the full load 50% of the time it would take to load the last chunk from its source**

NETWORKING – connecting to a source in a multi-source environment

In communications there are situations where a client needs to establish a connection with one of multiple sources, where the client does not care which of these sources it will connect to based on the information that it has prior to the connection. Such a case is may be a client that wishes to connect to a WiFi network where multiple such networks are available, or a cellular device that may connect to one of several antennas that have similar reception, or a client that wishes to connect to one of many DNS servers that are available to it. Many other such examples exist.

The connection between the client and the source typically includes a 'handshake' which is a series of interactions that are required to establish connection before the data transfer begins. For example, in TCPIP the following handshake states exist (from Wikipedia):

- LISTENING : In case of a server, waiting for a connection request from any remote client.
- SYN-SENT : waiting for the remote peer to send back a TCP segment with the SYN and ACK flags set. ('SYN-SENT' state is usually set by TCP clients)
- SYN-RECEIVED : waiting for the remote peer to send back an acknowledgment after having sent back a connection acknowledgment to the remote peer. ('SYN-RECEIVED' state is usually set by TCP servers)
- ESTABLISHED : The port is ready to receive/send data from/to the remote peer.

In some cases, this handshake may fail at one of the phases, in which case the client may choose to re-start the handshake process with this source, or alternatively may decide to start the handshake process with a different source.

The handshake process, which is typically defined in a standard, will typically have "timeouts" defined for each of the phases, after which time the handshake process is considered a failure.

Figure 200 is a diagram of a multi-source environment, where the Client 2 needs to connect to one of the available sources – Source 1 2004, or source 2 2002 or source 3 2006.

In such a multi-source environment, it is important to both select the source which has the highest probability to finish the handshake faster than the other available sources, and to cancel the handshake and start the process with a different source, in case it is anticipated that such cancelling and re-starting with another source will create a faster time to connection.

It is also possible that following each such handshake, the client communicates with a central server, and provides to that server information on the connection to the source, namely the time it took for each stage of the handshake. The central server stores this information and provides it to other clients prior to their connecting to sources, so that they can anticipate the time for each stage in the handshake.

Figure 202 is a flowchart that shows a sample implementation of minizing the handshake connection time. In 20202 the program receives the list of sources that the client can connect to, and numbers them S1…SN where S1 is the first source, and SN is that last of N sources available. The current source is set 20204 to the first source. Next, the information about the source is retrieved 20206 either from a local database within the client (based on the client's

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 35 of 85

previous connections to the source) or from a database on the network that keeps track of multiple client's connections to sources and shares this information between the sources.

There are multiple methods by which the connection times for each stage in the database may be calculated. 20208 shows how the database will store connection times for each stage: for each stage the connection time can be (i) the mean average between all past connection times, or (ii) the times for which 90% of the connection times fall under (or other such number), or (iii) the average times of the past X samples, etc.

In 20210, if the database does not include information about the source, the average connection time for each phase may be calculated as the average of other sources in the vicinity, or other sources within the database, or as the standard default time for the stage as is common in the market (typically defined within the specification of the protocol being used).

In 20212 N is set to 1, and in 20214 the timer for the current stage is set to the value retrieved in 20208 for this stage (source_t_stage_stagenum). In 20216 the actual stage in the handshake is done vs. the source, while allowing 20218 TIMER seconds to elapse for this phase (i.e. within TIMER seconds, either the phase in the handshake succeeds, or is expired). In 20220 it is checked whether the timer expired. If it has then in 20222 it is checked whether there are other sources to try to connect to. If there are more sources to connect to, then in 20224 the source to try is set to the next source in the list, and control resumes to stage 20206. If there are no more sources, then the process starts again with the first source, where in each stage the algorithm allows for more time per stage, by adding P% more time to each stage, where P may be set to 20% for example. If the stage succesffuly connected within the allocated time, then in 20230 it is checked whether this was the last stage in the connection. If it is, then the connection has been completed 20228 and the data transfer may begin. If it is not the last stage, then stagenum is incremented by 1, and the next phase in the handshake resumes in block 20214.

CLAIM:

1. **A system that comprises of a client that needs to connect to one of multiple sources via a process that has several stages in its handshake, where the client tries to connect to the source that it anticipates the handshake will take the least amount of time**
2. **In a system that comprised of a client that needs to connect to one of multiple sources via a process that has several stages in its handshake, a method by which the client anticipates the time of each stage of the handshake, and if the actual time for that stage exceeds the anticipated time by a certain factor, then the client cancels the handshake and tries to connect to a different source**

3. **The method of claim 2, where the anticipated time for each stage is the average of a sample of the times of previous connections to this source for this stage**
4. **The method of claim 3, where the anticipated time for each stage is provided either by a local cache within the client, and/or by a central server that collects previous connection times for the phases by communicating with the clients**
5. **The method of claim 2 where the factor by which the time is exceeded is the total average time of the handshake to this source**
6. **The method of claim 2, where the information of the actual times of stages in the handshake are stored on a central server, and are provided to clients before their connection for evaluating the time of each phase of the handshake, and are reported back to the server by the client after the handshake is completed or failed**

Heuristics for finding BW and RTT to network elements

In a network where a client may connect to one of multiple sources for fetching data as described for example in Fig 200, it is beneficial for the client to estimate in advance the Bandwidth (BW) and Round trip time (RTT) to each of the sources, in order to assess to which source it would be best to communicate with. For example, in hola's patent application number #12836059 , a client may have several peers from which it may load a chunk of data that it needs. That chunk may be for example 16KB of data. Assuming that there are two Peers (P1 and P2) with the following BW/RTT times – P1_BW=2,000 kb/s P1_RTT=30ms and P2_BW=4,000 kb/s P2_RTT=70ms, then the time for the transaction with P1 would be 30ms + 16,000*8/2,000,000 = 30ms + 64ms = 94ms, whereas the transaction with P2 would be 60ms + 16,000*8/4,000,000 = 70ms + 32ms = 102ms, in which case it would be more beneficial for the Client to communicate with P1. Other examples of such networks are when a client for HTTP data has 2 different web servers it can access for a certain URL (for example, the original web server and a CDN with the data).

However, the Client may not always have previously (or lately) communicated with the source that it needs to evaluate the BW/RTT with, in which case it would be beneficial to have an algorithm for evaluating the future BT/RTT of the communication with that element.

Figure 208 is a diagram that shows a network that includes a Client 2 and 4 sources (20802, 20804, 20806, 20808) and a database 20810 within the client that keeps track of the IPs, BW and RTTs of the various sources that it has communicated with in the past. This database may also include the time at which the last connection was made, as well as other data. Notice that this database has sorted the sources according to their IPs, such that if source_1 is in the

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 37 of 85

database before source_2, then necessarily the IP of source_1 is smaller than the IP of source_2.

In the example in Figure 208, the Client 2 has previously connected with source_1 20802 and with source_4 20808 and so has in its database 20810 the BW and RTT for these two sources. The Client 2 now needs to communicate with source 2 20804, and needs to evaluate the BW and RTT to source_2 20804, possibly because it has the alternative to seek a different source (or not to communicate at all) if the estimated time for the transaction would be longer than beneficial.

In such a case, the client needs to assess the BW and RTT of source_2. A good guess for the client would be to assume that the BW and RTT of source_2 are somewhere between the BW and RTT of source_1 and source_4, which are the sources of either side of source_2 (in terms of IP address) that the database is aware of. The exact BW and RTT that the Client can assume can be derived in various ways. For example, if the BW and RTT of source 1 are BW1 and RTT1 and for source 4 they are BW4 and RTT4, and for source 2 they are BW2 and RTT2, then BW2 and RTT2 can be calculated by their relative IP distance from BW1 and BW4 and between RTT1 and RTT4. Other ways to calculate can be a regular average (i.e. BW2 = (BW1+BW4)/2, etc.).

The database described in 20810 may also reside in a server on the network, and thus each client would be able to get information about connection to sources that it has not previously or lately connected with, because other clients have communicated with it and logged their results with this networked database.

Figure 210 is a flowchart of this process. In 21002 the client is maintaining a database of network elements that it has communicated with, where for each of these network elements it is keeping the BW and RTT of the connection, and optionally the time of connection for each. In the database, the network elements are sorted by their IP distance from the client, where the IP distance is the difference between the IP of the source and the IP of the client. The client is requesting 21004 to communicate with a source (source_x) and thus needs to first asses the BW and RTT from the client to source_x. In 21012 it is checked whether relevant information about the BW and RTT to source_x exist in the database (note that this database can be local on the client, or on a network and accessible by multiple clients). If it exists, then these are the BW and RTT that are assumed for source_x. If not, then the client gets information from the database about the BW and RTT for the two sources who's IP addresses are on either side (i.e. one lower and one higher) of the source in terms of IP distance. The client uses these two data points to assess the BW and RTT. This can be done in a variety of ways that are known by those that are skilled in the art, such as a regular average, etc.

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 38 of 85

For the above to work, much data about the connections to various network elements needs to be stored. Over time, this can accumulate to a large database, and thus a method that could reduce the size of the information while not giving up too much valuable information would be desirable. Figure 212 is a flow chart that shows one method in which this can be done by storing only data to certain elements which are indicative of elements in their vicinity.

In 21202, the system is initialized. This happens only when the system starts for the first time, before the database is initialized. In the initialization process, a database is created, with entries that signify logarithmic IP distances from the local host in base 2. Thus, the first entry will be for IP_dist=1, the second for IP_dist=2, the third for IP_dist=4, the fourth for IP_dist=8, etc., until the farthest IP distance possible in the network that the host is operating in (for example, in an IPv4 network, where the smallest IP address is 0.0.0.0 and the largest address is 255.255.255.255, the largest IP distance is half of the difference between these two addresses).

In 21204, it is checked whether the database is accessed for reading or writing. The database is accessed for reading when a program requests the estimated BW & RTT between the local host (who's IP is HOST_IP) and a certain network element to which this local host is considering to communicate with (who's IP is IP_NEW). The database is accessed for writing when a program finished communicating with a network element (or found out information about the BW & RTT to that network element in other ways), and requests to write the newly learned data (NEW_BW, NEW_RTT) to the database so that the system can make better assessments of the BW & RTT in future queries to it.

Next, the IP distance (IP_DISTANCE) is calculated 21206 between the HOST_IP to the IP_NEW by finding the number of IPs that are between the HOST_IP and the IP_NEW. Note that in the case of IP distance, this method considers that the IP addresses are 'connected at the edges', meaning that the first address (0.0.0.0 in IPv4) is 1 IP address away from the last address (255.255.255.255 in IPv4), and thus there are 2 different IP distances between each 2 points. IP_DISTANCE is calculated as the minimum between these two distances.

In 21208 the logarithmic distance (DIST) from HOST_IP to the NEW_IP is calculated. DIST is calculated by the following: Round_down(log_b2(IP_DISTANCE) – i.e. the rounding down of the log (in base 2) of the IP_DISTANCE between HOST_IP and IP_NEW.

If the database action was for a READ, then in 21212 the BW & RTT are read from the entry [DIST] of the database.

If the database action was for a WRITE, then in 21214 the NEW_BW & NEW_RTT are written to the entry [DIST]. Note that the NEW_BW & NEW_RTT may also be written to the database in other methods so as to keep track also of historical data. For example, the NEW_BW &

NEW_RTT may be averaged in with the other data samples (by keeping the latest average and number of samples in the database), or in other similar methods.

When network elements use BW & RTT tables, they could benefit from 'teaching' each other about the information they already have about BW & RTT between them and other network elements. Figure 214 is a flowchart showing a method by which network elements can share BW & RTT information so that a host that wishes to communicate with a network element that it has not recently communicate with can assess the BW & RTT to it by learning from other network elements that have recently communicated with that network element.

The method starts with scheduling an update of other network elements in 21404. The scheduling can be set to either happen at constant time intervals, or when the local host is idle, or when both the local host and other network element to be updated are idle, or when finished communicating with another network element, etc.

If the scheduled event of updating other network elements with the table from the local host, then a list is created 21408 of which network elements to update. This can be all network elements that have been recently communicated with, or a central server that will be updated and update other network elements, or by getting a list from such a central server, etc. The BW & RTT of this local host is then communicated to the elements in this list.

In 21410 the local host receives a BW & RTT table (UPDATE_DB) from a different network element who's IP is IP_ELEMENT. Then, for each non-empty entry in the UPDATE_DB (entry marked as E), the following set of actions is performed: First, in 21414 the IP of the element that provided the UPDATE_DB (IP_ELEMENT) is looked up in the local host's database, and the resulting BW and RTT are stored in memory as BW_ELEMENT and RTT_ELEMENT (these are the BW and RTT between the local host and the network element providing the UPDATE_DB). Then, in 21416, the BW & RTT in entry E are stored in memory as BW_E and RTT_E. In 21418 the IP distance from the network element to the IP in E is calculated and stored in memory as IP_DIST_E. in 21420 the local host's database entry for the IP distance of (IP_ELEMENT_DIST + IP_DIST_E) is updated where the BW value receives the value of BW_HOST + BW_E and the RTT value receives the value of RTT_HOST + RTT_E.

CLAIMS:

1. **In a system where a client needs to do a data transaction with a source on the network, a method by which the client may assess prior to the transaction the time it would take for the transaction, by estimating the bandwidth and round trip time to the source and make the decision whether to make the data transaction based on these parameters**

2. **The method of claim 1, where the estimation of the BW and RTT are done by using a database that contains previous connections to sources by which are logged in it**
3. **The method of claim 2, where if a source is not included in the list of sources, then a the BW and RTT of the source are estimated to be between the BW and RTT of the 2 sources that are on either side of it in an IP distance sorted list within the database**
4. **The method of claim 2, where if the client IP is not included in the database, then the BW and RTT are estimated as the average of the BW and RTT from the 2 sources that are on either side of the client in the IP sorted list within the database, to the source**
5. **The method of claim 2, where the database holds entries that represent IP distances that are of exponential distance from the local host where when new data about BW and RTT of a network element are added to the database, it is added to the entry which represents the IP closest to the IP of the network element, and when the database is queried about the BW and RTT of a network element it returns the information from the entry which represents the IP closest to the IP of the network element**
6. **The method of claim 5, where the exponent is of base 2**
7. **A method by which network elements which keep track of BW & RTT from themselves to other network elements occasionally update each other about the information they've collected about the network, so that elements have BW & RTT times from themselves to elements that they've not communicated with recently**
8. **The method of claim 7, where the BW & RTT table that is provided to a host by a network element is adapted by the host where each IP_DISTANCE in the table is adapted to be the IP_DISTANCE between the host and the network element, and where the BW & RTT are adapted by adding or reducing the BW & RTT from the host to the network element as the case may be according to whether the distance is shortened or lengthened**

GEO IP – creating contiguous range, optimal for speed

On the internet, regions are assigned IP ranges by the internet assigned numbers authority (IANA). More information about this is available at http://www.iana.org.

Neighboring regions will not necessarily have contiguous IP ranges between them (i.e. the end of one region will not necessarily be the beginning of its neighbor).

Distance between two IPs is typically considered the number of IPs between them. However, a broader distance between IPs may be use-specific. For example, for some uses, it may be beneficial to know the physical distance between two IPs (i.e. the physical distance between

the devices of the two IPs). In other uses, the distance between two IPs may be defined as the speed of communications between these two IPs.

Thus, creating one contiguous range for all IP addresses, based on the use, is beneficial. for example, for communication acceleration applications possibly having a contiguous range based on communication infrastructure and physical proximity, and for media caching applications by joining together regions that have similar cultures and/or languages.

For this, a 'cost' function needs to be determined, and the distance between IPs will be measured with this cost function, and thus a list of IPs by such order may be created. For example, the cost function may be the physical distance of the IPs, or the speed between them, etc.

Figure 216 is a flowchart for creating a contiguous geo IP range which is optimal for speed. In 21602 the cost function is determined (COST), and in 21604 this cost function is used to create a contiguous list of any N IPs (for example all IPs in the Internet), where ∑0 TO N COST(P(I),P(I+1)) == MIN, i.e. where the sum of the costs of all neighboring IPs in the list is a minimum vs. all such similar lists.

An example use for such a list is a list of all IPs on the Internet, where if IP1 and IP2 are closer to each other than IP1 and IP3, then by necessity the communication between IP1 and IP2 is faster. This can be because they are geographically closer, or because they belong to the same ISP and thus their communication would be faster. For example, if the cost function is to sort IPs by their belonging to the same network service provider, then that will be a reasonable approximation for the speed as a cost function.

Finding out which elements in a networked system are online

Figure 218 is a diagram of a network in which there is a central server 21804 which may communicate with several network elements 21802. There are cases, such as in Hola's patent application number #12836059 where it is beneficial for the server to know which of the network elements 21802 are online. Polling the various elements is costly in terms of bandwidth and computational power, and therefore more efficient ways of finding out which elements are online may be beneficial.

One way to figure out if the element is online is for the server 21804 to keep a database to keep track of the network elements that are online, and to update the status of each of the the network elements by having the element 'ping' the server every X seconds, where X is a pre-set number in the system (for example every 10 seconds).

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 42 of 85

Figure 220 is a flowchart that shows an implementation within a network element and a central server by which they interact so that the server keeps an updated list of online elements.

In 22002, the operation of the network element is described: every preset time of PING_CYCLE_TIME seconds (could be 10 seconds for example), the network element pings the server.

In 22004, upon initialization of the server, the database is created with an entry that corresponds to each network element in the network, as well as two Boolean fields: 'ONLINE' – which will indicate whether that network element is online, and 'UPDATED' which will indicate whether that network element has pinged the server since the last time the server checked the database.

Every time a ping is received 21406 by the server, the server updates the entry in the servers' database that corresponds to that network element to ONLINE (ONLINE=TRUE) and marks its UPDATED field as TRUE.

Every time that PING_CYCLE_TIME seconds pass within the server, it performs a review of the database; for each entry of the database that is marked as ONLINE (ONLINE=TRUE), it checks whether its UPDATED=TRUE. If it is not, it marks the entry as not being ONLINE (ONLINE=FALSE). If it is, marks UPDATED=FALSE.

When the server is queried on whether a network element is online, it looks up the database entry that corresponds to that network element, and returns the value in the ONLINE field (TRUE or FALSE). TRUE= network element is online, and FALSE=network element is not online.

CLAIMS:

1. **In a system where there are multiple network elements that may be online or offline, and a central server that can communicate with the network elements, a method for the server to keep track of which network elements are online, by which the network elements ping the server every X seconds when they are online, and the server marks in its database which elements have pinged it within every similar time period of X seconds, and at the end of that period marks all elements that have not pinged it as being offline, and those that have pinged it as being online**

Dynamic queues for sending data

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 43 of 85

Figure 224 is a diagram showing the prior art in how applications 8 within a local host 22402 communicate with network elements using the operating system services. Applications 8 that need to send data to network elements such as 22404 or 22406, they pass the data to be communicated to the operating system 10. The operating system places that data on an OS Queue (such as that in 22002). The data is kept by the operating system in the OS queue 22402 until the underlying socket 22404 is free to send the data to the network element through the local host's 22402 physical connection 22406. The physical connection to the network can be Ethernet, WiFi, USB, or any other connection. Often a host will have multiple physical connections 22406.

When data is sent to an os queue (like 22402), it is committed to that queue until it is moved to the socket. During the time that the data is kept in the OS queue, a different OS queue could become empty (e.g. because it has finished sending the data to the network). In such a case it would be desirable to move the data from its current OS queue 22402 to the OS queue of the socket that is free. However, it is not possible to move the data between OS queues, and thus the data is not always sent to the network as quickly as it could be.

It is thus useful to create a system or a method that would allow to always send the data from an available socket. Figure 226 is a diagram of a system that makes it possible to move data between sockets so that if data is waiting to be sent through a socket and during that time a different socket is ready for sending data, the data is moved to that other socket.

22602 is a host in which applications 8 are running and using the services of an OS 10, that includes OS queues 22608 as described previously, that are associated with sockets 22610 and physical connections 22612. A new module is added to this system, called the 'dynamic queue module'. This module allocates Dynamic queues 22606 that are associated with the OS queues 22608, so that when data is being sent to the network by the applications 8, the dynamic queue module allocates the data to the dynamic queues associated with the various sockets. While the data is queued in the dynamic queue 22606, if its underlying socket is ready to send data, then the dynamic queue 22606 sends that data to the OS queue 22608, and from there the OS moves the data to the socket 22610 to be sent through the physical connection. However, if while the data is queued in the dynamic queue 22606 the dynamic queue module learns that a different socket is ready for sending data to the network, the dynamic queue module removes the data from its current dynamic queue 22606 and moves it to the dynamic queue that is associated with the socket that is now ready to send data to the network.

Figure 228 is a flowchart for the dynamic queue management. In 22802 an application queues data of size DATA_SIZE to be sent to a communication device. The data is intercepted 22804 by

the dynamic queue module. This interception may be done in various ways that are known to those that are skilled in the art, such as by hooking which is described in http://en.wikipedia.org/wiki/Hooking.

In 22806 a list is created of all the sockets in the system that the data may be queued on. For each such socket in the list, store in memory the amount of data queued in the dynamic queue and the OS queue (WAITING_DATA_SIZE) as well as the BW/RTT (SPEED). In 22810 the WAITING_DATA_SIZE and SPEED are used to determine which socket to queue the new data on. There are multiple methods of deciding which socket to queue the data on, based on the DATA_SIZE, WAITING_DATA_SIZE and SPEED. These methods would be obvious to those skilled in the art, for example by calculating the estimated time it would take each socket to be freed (estimated by WATING_DATA / SPEED) and adding to that the estimated time it would take for that socket to transmit the DATA_SIZE (estimated by DATA_SIZE / SPEED). Once the system has chosen the socket that is the most likely to be free for transmitting data first, it queues the data on the waiting queue associated with that socket. In 22812 it is checked whether the socket that is underlying each dynamic queue (DQ) is empty. If it is, then in 22814 the next data from the DQ is queued to the socket (typically to the os queue), and removed from the DQ. If not, then it is checked whether any data (D) that is currently scheduled in a DQ has been cancelled by the application that requested to send it. If so, then D is removed from DQ. If not, then in 22820 it is checked whether there is a different dynamic queue (DDQ) that is now more optimal for D than its current dynamic queue (CDQ). If there is, then D is moved 22822 from CDQ to DDQ.

CLAIMS:

1. **In a system that includes one or more application that require sending data to network elements through the operating system, and where the operating system queues these messages for sending through sockets, a method by which if data is queued for being sent through a socket, and before it is sent a different socket is ready for sending data, the data is moved from its queue to the socket which is ready**
2. **The method in claim 1, where a layer of dynamic queues is added to the system, where the data that is to be queued to a socket through an OS queue is instead queued in a dynamic queue that is associated to the OS queue, and is only inserted in to the OS queue once the underlying socket is ready for sending data, and if while the data is waiting in the dynamic queue and a different socket is ready for sending data, the data is un-queued from its dynamic queue and sent through the socket ready for sending**

3. **The method in claim 1, where if the sending application requests to cancel the data it requested to send that is still pending to be sent, that data is removed from the socket's queue**

Virtual application gateway

A network element that wishes to communicate on a network will typically identify its gateway in the network, and from it will obtain its IP address, DNS server's IP, its subnet mask, and other information that it will use during the communication process.

Figure 232 is a diagram that describes that prior art of how a network element 23202 which has an operating system 10 and applications 8 running on it, is joining a network 6, and receiving configuration information 23206 from a gateway 20.

This gateway could be a network router or a WiFi router, or any other gateway device. If the network element 23202 is disconnected for any reason from the gateway, and then re-connected to the gateway or a different gateway, the applications 8 lose connectivity during the time that the communications device 8 was not connected to any gateway. This has some negative implications, such as disruptions in service (e.g. "404 page not found" in the browser – see http://en.wikipedia.org/wiki/HTTP_404), re-configuration taking some time, etc. It is thus useful to have a different architecture and method by which such disconnects have a lesser impact on the network element 23402.

Figure 234 is a diagram that shows such an architecture. A network element 23402 has an operating system 10 and applications 8, with a virtual application gateway module 23404 running on it as well. When the network element 23402 is connected to a gateway 23410, the virtual application gateway 23404 intercepts the operating system's request to the network for receiving configuration information, and receives on its own the configuration information from the gateway 23410 and provides that information back to the operating system, thereby acting as a sort of proxy for the initial configuration information. Some of the requests that the virtual application gateway module 23404 receives may require it to do actions on its own which are not in the form of being a proxy. One such action is the IP request from the operating system – in that case the virtual application gateway module 23404 will need to provide a local IP to the operating system.

In this system, when the network element is disconnected from gateway 1 23410, and then reconnected, or connected to a gateway 2 a short time later, the virtual application gateway module 23404 can simulate to the operating system that there has not been a disconnect by not reporting the change of status. When the re-connection to a gateway has been done, the virtual application gateway module requests new configuration information, but does not need

to change any of the information that it has passed on to the operating system. Thus, the operating system from its perspective was continually connected to a network, although in practice the network may have been disconnected for a time.

Note that if the network element is disconnected from a gateway for a period of time which is longer than some pre-set length of time, the virtual application gateway module will report this to the operating system, so that the operating system 10 and applications 8 have the ability to respond correctly to the situation (e.g. notify the user, etc.).

Figure 236 is a flowchart of how such a virtual gateway may be implemented. As the network element is connected 23602 to the network, the virtual application gateway module VAGM requests an IP from the gateway on the network, that the OS would have requested an IP from. Then, the VAGM sets 23604 itself up for intercepting the IP requests of the operating system that would normally have gone out to the gateway. This type of interception is may be done in a multitude of ways that would be obvious to one skilled in the art. The VAGM then waits to intercept 23608 an IP request from the OS to the network. Once such a request has been trapped 23610 the VAGM provides an IP to the OS in a similar way to how NAT devices provide IPs to requesting clients. The VAGM then acts 23606 as a proxy/nat for the operating system as NAT devices do. The VAGM then checks 23612 whether it has disconnected from the network. if not, it continues with 23608. If it has been disconnected from the network, then it tries to get an external IP from the local gateway (if it is now connected again to the network), or simply skips this step if it is not connected to a local network any longer. In 23616 the VAGM checks whether it has received an IP from the network. If it has, then it continues with 23608. If it has not, then it allows up to x milliseconds to be without connectivity. If that time has passed 23620 then the VAGM sends a signal to the operating system that the connectivity is lost (for example by disconnecting itself from the operating system). This allows for the operating system to react to a disconnect in the way it was programmed to. If it has not exceeded the X milli seconds, then it checks again whether it has received an IP from the gateway, and if so continues normally with being a NAT/proxy for the OS to the network.

CLAIMS:

[4:30pm]

1. **In a computing device that is connected to a network and includes an operating system and applications, a method by which a program on the device provides the communication configuration to the operating system instead of the operating system getting it from a gateway on the network, where the program communicates with the external gateway to get the configuration information so that it can communicate to other network elements, and provides separate configuration information to the**

operating system, thereby having the operating system communicate with the program, and the program communicating with the external network elements

Reliable connection proxy

Connections on the network have a reliability that is typically not 100%. The reliability is typically measured in the number of packets that don't reach their destination in tact, but can be measured by their latency, bandwidth and other factors. Figure 240 is a diagram of prior art, where network element 1 (NE1) 25 wishes to communicate with network element 2 (NE2) 26, where there are multiple unreliable connections between the two elements. However, in the prior art, a connection for a specific piece of information that NE1 25 needs to fetch from NE2 26 is established over only one of these unreliable connections. For example, if NE1 25 is doing a voice over IP call to NE2 26 over one of these unreliable connections, which for the sake of the example drop 1% of the packets, then 1 out of every 100 packets of the call will be lost, and the call's quality will be lower. If the other unreliable connections between the two network elements could be used to increase the reliability, that would be beneficial.

Figures 242, 243 and 244 is a diagram of how the additional connections may be utilized for increasing the reliability of the connection.

In figure 242, NE1 25 wishes to communicate with NE2, and wishes to utilize the multiple unreliable connections 24204 between them. For this purpose, a special program is installed on both NE1 25 and NE2 26. This program causes each packet that is transmitted from NE1 25 to NE2 26 or vice versa, to be transmitted over two or more routes. Thus, the unreliability of the connection is decreased to a factor received by multiplying the unreliability of the routes, and the bandwidth that is used is increased to a factor which is the number of routes used. So for example, if 3 routes are used, where the reliability of each is 99% (i.e. 99 out of 100 packets typically reach their destination in tact), then the unreliability of the current connection is 1%, and the unreliability of the new method of communication is 1% (unreliability) ^ 3 (number of routes) = 0.0001%, and the bandwidth that is used is 3 times the bandwidth used in the state of the art method since each packet is transmitted over 3 routes. Using multiple routes may be done in various ways that are known to those skilled in the art, for example by using a program that uses the network element's multiple network interfaces and sends the packet over each of those interfaces, or by using peers in the network as proxies, such as with hola's patent application number #12836059 .

There are cases in which the program is installed on one network element, but not on the network element that is being communicated with. In such a case it may be beneficial to communicate via a reliable proxy on which the program is installed. If this proxy has a more reliable connection to the other network element, than this can be beneficial. For example,

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 48 of 85

Figure 243 is a diagram where NE1 25 wishes to communicate with NE2 26. The program is installed on NE1 but not on NE2. Therefore, NE1 queries a reliability proxy network server 24301. It provides to the server the identifier (e.g. IP) of NE2, and receives the identifier (e.g. IP) of the reliability proxy (RP) 24308 to use. This RP 24308 is a network device on which the program is installed, and which has a connection 24310 to NE226 which is more reliable than the direct connection between NE1 25 and NE2 26. NE1 25 then communicates with RP 24308 via multiple routes 24306, and RP 24308 communicates with NE2 26 through the more reliable connection 24310. Thus, 24306 is a more reliable connection than the one route available between NE1 25 and NE2 26 (because the original route is included in 24306, and other routes improve its reliability), and 24310 is more reliable by selection, and thus the total reliability is higher than a direct connection between the two devices.

It is also possible to use this method in a configuration where both NE1 25 and NE2 26 do not have the program installed on them. Figure 244 describes such a case, where NE1 25 is manually configured to use the reliability proxy 24404 as a proxy, and that proxy then uses the reliability proxy network server 24401 to find the reliability proxy 24406 and to use it to communicate with NE2 26.

Figure 246 is a flowchart describing how such a program may be implemented. The program is activated 24602 when NE1 needs to communicate with NE2. The program checks 24604 whether the existing direct connection between the two is good enough. This can be application specific – e.g. if the existing bandwidth and round trip are enough for the needs of the communicating application. If the existing route is good enough, then a direct communication is done 24606 as in the prior art. However, if the existing route is not good enough, then in 24622 the program checks whether NE2 has the program installed. This can be done by sending a query to NE2 that if the program exists, this query would be returned positively, or can be done by checking with the reliability proxy network server by asking it whether NE2 has the program installed (the elements on which the program is installed should report to the server when they are online, and the server should log this in its database). If NE2 does have the program installed, then a direct communication is done between NE1 and NE2, via multiple routes. This process is described in figure 247. If NE2 does not have the program installed, then a request is sent 24608 to the reliability proxy network server (RPNS) for a proxy to use for the communication. The response received 25610 from the RPNS provides a reliability proxy device (RPD) who's path to NE2 is more reliable than a direct connection from NE1 to NE2 if such a path exists (if there is no such path, then direct, prior art type communication is established, such as in box 24606). NE1 then sends 24612 the information to RPD via the multiple routes between them as described in figure 247.

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 49 of 85

In figure 247, a method of transmitting packets in parallel over multiple routes is described, in which one communications device (NE1) wishes to communicate with another communications device (NE2), sending the same data over multiple routes in parallel. In 24702 NE1 gets from the routing unit the number of routes available from it to NE2, and the reliability of each one. In 24704, the routes to use are determined by doing the following: the desired reliability that is requested from the requesting application is set as DR. The available routes are ordered by their reliability, where the highest reliability is the first route in the list - route 1, with a reliability of R1, the second is route 2 with a reliability of R2, and the last is Route N with a reliability of RN. The routes to use are determined by multiplying (1-R1) * (1-R2) * (1-R3) …, until the product is lower than DR. Note that (1-Rn) is the "un reliability" of route n whose reliability is Rn, and thus the product of the 'un reliability' of the routes is the un-reliability of the parallel use of routes. Thus, assume that the routes R1, R2, … Rx are used for this particular communication between NE1 and NE2. During the communication process between NE1 and NE2, for each packet to be transmitted, transmit it 24706 over all chosen routes. On the receiving side, for each unique packet of communication received, send it 24708 to the application, and discard the other similar packets received from the other routes (since they have already been received). In the case of a missing packet over the fastest connection, that packet will have been received from the second fastest connection.

Aside from packet loss, another problem of unreliable connections is that they may disconnect from time to time. When they disconnect, the communications devices on both sides will try to reconnect, and will often succeed doing so. However, the disconnection creates in-continuity for the communicating applications (e.g. a "404 page not found" in a browser), whereas if the disconnection was not signaled to the application until the re-connection is done, this in-continuity would not have occurred. Thus it is desirable to have a method of delaying the signal to the applications for a short time that is required to re-establish the connection if that is possible.

Figure 248 describes such a method. In 24802 a network element NE1 wishes to communicate with another network element NE2, and wishes to keep the connection continuous if possible. The connection from NE1 to NE2 is established 24804 through the reliability proxy program (RP) in each of NE1 and NE2 if such a RP exist in them. If the RP does not exist in NE1, then it can be configured to use an external device as a reliability proxy (RP). If the RP does not exist in NE2, then NE1 requests an RP from the reliability proxy network server, and communicates with it instead of with NE2, where the RP will proxy the messages to NE2 so that effectively NE1 will be communicating with NE2.

In 24810 it is checked whether the connection between the RPs has disconnected (whether the RPs are within NE1 and NE2 or external to them). If the connection has disconnected, then in

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 50 of 85

24812 the reliability proxies hold the connection between them and the operating system that they are acting as a proxy for, for a set short amount of time that it would take to re-connect the broken connection in a reasonable scenario (e.g. 200ms). Holding the connection is done where a system includes a solution similar to the virtual application gateway described in figure 236. During the time that the connection to the operating system is held, the RPs try to reconnect 24814 with each other. If the re-connection succeeded within the set time, then communication resumes without a break from the operating system's perspective. However, if the re-connection did not succeed in the set amount of time, then the OS is notified of the disconnection.

CLAIMS:

1. **In a system that comprises of two communications devices that are communicating with each other, and having more than one possible route between them, a method of communication by which each segment of information is communicated in the more than one route concurrently**
2. **In a system that comprises of two communications devices having more than one possible route between them, that are communicating with each other, and where one or more of these devices do not have the ability to communicate via these more than one possible routes, having a reliability proxy server, and one or more reliability proxies, a method by which the communicating device requests from the server a proxy that has a more reliable connection to the other network element, and communicates with that proxy through the more than one routes, and the proxy communicates that information with the other network element through the more reliable connection they have between each other**
3. **The method of claim 2 where the network element initiating the communication has a reliability proxy set as its communication proxy so that all its communications go out to the proxy, and the proxy then uses the multiple routes to communicate with the other network element or with a reliability proxy that has a more reliable connection with the network element.**
4. **a method of communication by which each segment of information is communicated in the more than one route concurrently**
5. **[this claim is for the disconnection reliability proxies] in a system that includes network devices, some of which include reliability proxies within them, a method by which where a network device wishes to communicate with another communication device and for the connection to stay continuous as much as possible, where the connection between the devices is done through reliability proxies that can be embedded in the communication devices, or external to them, received from a central reliability proxy server, so that if the connection is disconnected, the reliability proxies do not notify the operating system that they are a proxy for while trying to reconnect to the other communication device**

6.

Aggressive DHCP

DHCP is the process by which a communications device receives the IP to use when communicating with other communications devices. DHCP is further explained in http://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol.

Figure 260 is a diagram of how DHCP servers 26004 and 26006 may typically be connected to a host communication device 26002. When the DHCP server is on the same LAN 30 as the host 26002, as is DHCP server 26004, the typical round trip time is around 5 ms. However, when the DHCP server is on the WAN 6 as is DHCP server 26006, then the typical round trip time is around 20ms.

Figure 262 describes the prior art DHCP. In the prior art, when a DHCP request was sent by a host to a DHCP server, a countdown timer is set 26202 (to typically approximately 5 seconds), the host sends out 26204 the DHCP request to the DHCP server, the countdown timer is started 26206, and then it is checked 26208 whether the response from the DHCP server is received before the countdown timer has expired. If it has, then this answer is used. If not, a DHCP timeout is signaled to the system, and the system can consider its actions, which can be for example to try to connect to a different DHCP server, or a different network on which it hopes to receive a DHCP response.

Note however, that the timeout that is typically used (around 5 seconds), is orders of magnitude longer than the 5ms or 20ms round time to a typical DHCP server, and thus in case the host is communicating with a dysfunctional DHCP server, it will have to wait out the full 5 seconds to figure out that it is dysfunctional, whereas if the answer was not received within 500ms it would probably have been reasonable to assume that the DHCP is dysfunctional or that the request to the DHCP server was not properly received by the server (due to lossiness of the network, or reasons internal to the DHCP server or the requesting host).

Thus, it would be beneficial to introduce a new method to be more aggressive by providing shorter timeouts for DHCP requests, so that if the problem is that the message was not properly received, it will be retransmitted faster.

Figure 264 is a flowchart for shortening the average connection time to a router, by introducing more aggressive DHCP connection timeouts for the above reason.

When a DHCP request is about to be made, a countdown timer is set 26402 to a preselected time T which would be suitable for close by, fully functioning DHCP servers (for example T=5ms). The host then sends the DHCP request to the DHCP server in 26404, and then starts

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 52 of 85

26406 the countdown timer. In 26408 it is checked whether the answer has been received before the countdown timer has expired. If the answer was received 26410 then that answer is returned to the program that asked for the DHCP. However, if the timer expired, then in 26414 it is checked if T is equal to or larger than the DHCP timeout that would have been used in prior art implementations (e.g. 5 seconds). If it is, then return 26416 to the calling application with "no answer" (i.e. could not get DHCP response). If however the DHCP timeout has not been reached, then the countdown timer's time T is incremented (for example, it is doubled), and another DHCP request is sent 26404 to the DHCP server.

CLAIMS:

1. **A method of retransmitting DHCP requests by a requesting communication device to a DHCP server, by which a short timeout is defined, by which if the result from the DHCP server was not received within that timeout, then the timeout is incremented, the request is retransmitted, and so forth, until either the response has been received, or the timeout is larger than the prior art timeout for DHCP requests.**
2. **The method of claim 1, where the short timeout is defined as 5ms.**

Meta data prioritization

In a system such as Hola's patent application number #12836059 , network elements need to send data messages between them, and also often need to send meta data. Meta data is distinguished from the 'data messages' in that this is information that is less time sensitive, and in many cases may be completely ignored without materially impacting the performance of the system.

In such cases, the Meta data may sometimes interfere with the performance of the message data that is more time sensitive.

Figure 270 is a diagram of the prior art. Host 27010 is sending data messages 27002 to the network element 27020, and is also sending meta data messages 27004 to various elements on the network. The messages 27002 and 27004 are sent in the order that the host creates them and decides to send them on the network. This creates a situation where the messages sent 27006 are a mix of both types of messages. If there is a constraint on the communications channel between the host 27010 and the network element 27020, that is reached due to the traffic that both the data messages 27002 and the meta data messages 27004 are generating, then in effect the meta data messages are slowing down the delivery of the data messages (or otherwise effecting it).

Thus, a solution by which the meta messages do not impact the performance of the data messages would be beneficial.

Figure 272 is a diagram of such a solution. The host 27201 is sending data messages 27202 to a network element 27206 and meta data messages 27204 to other network elements. Both of these types of messages are routed through a Meta data routing unit (MDRU) 27220 which may be implemented within the host 27201, or on a separate device. The MDRU routes the data messages 27202 directly to the network element 27206. For the meta data messages 27204, the MDRU first stores the outbound meta data messages in a meta data queue 27208. It then checks whether the current traffic of data messages to the network element be adversely effected by sending the next meta data in the queue now. It does this by evaluating the data messages 27202 in the queue. If there are still regular data messages 27202 in the queue, then the meta data messages are not sent. If however the queue is empty, then the next meta data message in the queue is sent 27212. The MDRU may also asses other parameters when considering whether to send the meta data messages 27204. For example, it may evaluate whether the CPU of the host 27201 is busy, whether the disk of the host 27201 is busy, etc., and not transmit the meta data messages 27204 if they are, so that these already constrained resources are not used for the meta data messages 27204 which are not time sensitive.

claims:

1. In a system where a host communication device is communicating with other network elements and occasionally needs to send less time critical meta data to network elements, a method of prioritizing sending of the data by only sending the meta data after the network device has sent all outstanding data messages

COMPRESSION – compression with non deterministic shared dictionary

Many lossless compression algorithms rely on previously transmitted content to create a 'dictionary' (or table) of previously used strings of data. More about compression theory can be read at http://en.wikipedia.org/wiki/Data_compression and is incorporated herein in its entirety.

Compression can be done for various reasons, including to save storage space, or to save transmission capacity when data is communicated between networked devices.

When two communications devices communicate with each other and wish to compress their communication, a buffer is kept in both of the communications devices, and in it is stored a history of their communications and using that buffer as the 'dictionary' between the two devices. This dictionary is stored per communication channel – i.e. if Host A is communicating with Host B, then the transmission from Host A to Host B will generate and use one dictionary, and the communication back from Host B to Host A will generate and use a separate dictionary.

Figure 300 is a diagram of prior art. 30002, and 30010 are Networked communication devices (Host A and Host B respectively). Each of these has applications 30004 & 30012 running on it, where Host A is transmitting through TX (transmit) 30006 to Host B on RX (receive) 30014. Host A stores the messages

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 54 of 85

sent through TX_A in its shared dictionary 30008, and Host B stores the messages it received through RX_B in its shared dictionary 30016. The communication from TX_A 30006 to TX_B 30014 is compressed and decompressed using the shared dictionary which is identical on both machines.

As more communications are done, the larger and more meaningful this dictionary will become. We call this type of compression a "deterministic" compression because when Host B receives a compressed stream from Host A, it always is able to decompress this stream because of the joint dictionary they both share.

There is a likelihood that there is some commonality between the information stored on Host A 30002 and that which is stored on Host B 30010. For example, in Hola's patent application number #12836059 , this shared information may be similar information passing between the hosts, either in the data portion or the headers of the communication.

Thus, it would be beneficial to have a method by which hosts would be able to have a common dictionary which is not dependent only on the information they've directly shared but which is based on the shared information they may have in advance of the their communication with each other.

Obviously, when a new host is introduced in to a system that is employing this method, it would not be aware of the dictionary being used by these hosts. Therefore, this method can use a 'non deterministic' shared dictionary, which means that it will assume a dictionary that exists by all participating parties, but in some cases some of the hosts will not have parts of the dictionary being used in the compression, and thus will request more information from the transmitting host in order to be able to decompress the data.

Figure 301 is a diagram of compressing a stream between hosts with a non deterministic shared dictionary. Host A 30102 is transmitting information through TX_A 30108 to Host B 30120 who is receiving this information through RX_B 30126. Host A 30102 has a shared dictionary 30110 for the communication it transmits to Host B 30120, and in addition it has an "Other" dictionary 30112 which is a dictionary for communicating with all network devices in a non-deterministic manner as will be described herein. Host B 30120 has a shared dictionary 30128 for the communication it receives from Host A, and in addition it has an "other" dictionary 30130 similar in description to 30112.

The method of operation for this 'non deterministic' compression is that the communication between the hosts is done through a "non deterministic shared dictionary module" 30106 in Host A 30102 and 30124 in Host B 30120. When this module starts up, it collects information stored on its host machine, and stores it in fixed sizes ("Chunks"). For each such chunk it stores its checksum. During the course of the host's operation, the module continues to store additional information as it is added to the host's storage devices, or as it is received by the host (through its communication with various hosts for example). When Host A compresses information to send to Host B, it uses the shared dictionary 30110 and its 'other' 30112

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 55 of 85

dictionary to look for an entry that would enable the maximum compression (e.g. the smallest outcome, or least CPU cycles used) and compresses the information. When the information is received by Host B, it is decompressed using its shared dictionary 30128 and its 'other' dictionary 30130. If these dictionaries do not enable the decompression of the data, then it is because the data was compressed using information in Host A's 'other' dictionary 30112 which does not exist in Host B's 'other' dictionary 30130, and in such a case Host B will transmit a 'retransmit' message back to host A (marked as the Feedback 30130 channel in the diagram), and Host A will re-transmit the information compressed only with the shared dictionary (or completely uncompressed if there is no shared information that may help).

Figure 302 is a flowchart for how such a method may be implemented. First, as the system loads the buffer size for the local dictionary is determined. This can be done in a multitude of ways, for example by setting 30202 a hard coded number (e.g. setting aside a 1GB buffer for the local dictionary), or by using a portion of the available disk space, or other methods. Next, the local dictionary is built. This may be done in a variety of methods, including to simply scan all of the information stored on the various storage devices of the communication device, and keeping a list of the checksums of each of the chunks in the storage devices, and using these checksums as a the compressed version of the chunk itself. Next, a probability is allocated 30204 to each of these checksums, which expresses the probability of finding this piece of data in other communication devices. This may be done in a variety of heuristics. For example, allocating a high probability to any files that belong to the operating system (since these will be mostly identical in other machines with the same probability), low probability to files that were generated locally, etc.

When the local communication device (host) sends information to a remote communication device (target), the program splits 30208 the data being sent in to chunks, and for each such chunk it checks 30208 whether some of this information is available in its dictionary. If the data exists 30214 in the dictionary, then the host sends 30216 the data identifier of the dictionary data. If the data does not exist, then the host compresses 30212 the chunk via normal method (based only on the stream sent to the other host for example), and then moves 30218 to the next chunk. Once the host finished sending all this current data 30210 it returns to checking 30206 whether another message needs to be sent, or whether a message was received.

If a message was received, then it checks 30220 whether this is a new message received, or a retransmit request for a message already sent. If this is a 'retransmit', then the meaning of this is that the target did not have the assumed information in its dictionary needed in order to decompress this message, and thus the host transmits 30222 the message again using only normal method of compression. If a new message was received, then the host looks up 30224 the message identifier within its dictionary. If the message identifier was found, then it is used

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 56 of 85

to decompress the message (i.e. the data to which the message identifier in the dictionary points to, is the data used instead of the message identifier, and thus is the 'decompressed' data in itself). If the message identifier was not found, then a 'retransmit' message is sent 30230 to the target.

Figure 304 is a flowchart for implementing an improvement on this non deterministic shared dictionary. In the implementation offered above for the non deterministic shared dictionary, the data that is received from other hosts may be marked as shared data with those hosts, while other data that is received by scanning the local storage and from other hosts can be marked as 'other data'. Thus, when compressing data vs. a host, there is data that is shared between the transmitting device and the host, and there is 'other data' that may or may not be present with the host, and thus if used in the compression dictionary, this 'other data' may not be recognized by the host, and thus a 'retransmit' will be required, wasting bandwidth and time. The improvement is in that to each 'other' data, a probability is assigned as to whether the other host may have this data, to the best of the transmitting device's knowledge. This probability factor per data item may be done per chunk of data, or per file of data, or per URL, or any other data size. The probability may also be calculated as the probability that a random device may have that data, or per family of devices (e.g. probability that laptops will have it, probability that tablets will have it, probability that phones will have it, probability that Windows PCs will have it, etc.), or it can be done per host (i.e. the probability that 74.125.224.134 will have it, etc.

Now that probability per data exists, then the benefit of using that 'other data' in the shared dictionary may be calculated, by considering the bandwidth saved by compressing with this 'other data', vs. the probability of the other host not having this data and thus losing the original transmission time, plus the time it takes to get the 'retransmit' message from the other host. Thus, when the expectancy of using the other data is below a certain threshold (e.g. zero), then the 'other data' is not used for compression.

Figure 304 is the flowchart for this improvement. In 30402 some parameters are set as follows:

- BW = upstream bandwidth to destination
- segment_compressed = segment size when compressed with shared dictionary only
- segment_compressed_with_other = segment size when compressed with the 'other data'
- bits_reduced = segment_compressed – segment_compressed_with_other
- prob = estimated probability that destination has the relevant 'other data'
- penalty = (segment_compressed_with_other/bw + rtt) + rtt (the segment_compressed_with_other/bw + rtt is the time it takes to transmit the data to

the other host, and the rtt added to it is the time it takes the device to receive the 'retransmit' message sent by the other host in case it does not have the 'other data'

In 30406, the expectancy of the time saved is calculated:

EXP_T_SAVED = PROB * (BITS_REDUCED/BW+RTT) + (1-PROB) * PENALTY

Where exp_t_saved is the expectancy of the time saved by using the 'other data', and it equals the probability times the difference in bits between the messages (i.e. the time that is saved by compressing the message) plus the inverse probability (ie. The probability of the host not having that 'other data' times the penalty that would incur.

In 30404 it is checked whether this will save time by checking the expectancy calculated in 30406. If it will save time, then the data is transmitted 30408 after it is compressed with the 'other data'. Otherwise, the data is compressed only with the 'shared data'.

Another improvement in the non deterministic shared dictionary approach relates to the 'retransmission' of messages in the case that the other host does not have the 'other data' that was used to compress the message. In such a case, the non deterministic shared dictionary algorithm described above will retransmit the data that was compressed with the 'other data'. However, in some cases it may be more beneficial to transmit only that 'other data that was used for the compression, instead of the data that was compressed. This may be calculated in a variety of ways, such as considering what is the time it takes to for the host to receive the retransmitted message, vs. the time it takes for the host to receive the 'other data' required plus the time it takes that other host to decompress the message based on that new 'other data'.

Another improvement on this system is that there are times when the other side is missing information in its dictionary to to decompress the data in which it is 'cheaper' in terms of bandwidth or otherwise to transmit the missing information in the dictionary instead of transmitting the data itself.

For two way dictionary, use the following:

In many cases, when two hosts are communicating, much of the information transmitted from one host to the other has similarities with the responses received from the other host to the first. For example, in Figure 310 if Host A 31010 is a PC and Host B 31020 is a web server, and host A 31010 is fetching a web

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 58 of 85

page from Host B 31020, then the requests are being made from Tx_A 31012 to Rx_B 31024, and the responses are being made from Tx_B 31022 to Rx_A 31014, where each of these two channels will have their own dictionary (one dictionary for A->B communications, and one dictionary for B->A communications). However, there will be commonality between the information flowing on both of these channels, for example within the HTTP headers.

Figure 312 is a diagram of a system that performs two way compression with one joint dictionary. Host A 31210 communicates with Host B 31220, and vice versa, but instead of having two joint dictionaries, they have just one. The applications communicate with their respective Tx and Rx units (31216 and 31218 for Host A 31210, and 31220 and 31222 for Host B 31220). Instead of communicating with the network, these units communicate with a Context Module (context module A 31224 for Host A, and context module B f31226 for Host B). This context module monitors the information flowing from host A to B and from B to A, and maintains a shared database that can be used for the compression and decryption of the data streams.

Having a joint dictionary for both streams as opposed to the prior art of a shared dictionary per stream, reduces the bandwidth between the two communicating devices, at the expense of CPU power on both devices, which is increased when using such a system.

Figure 314 is a flow diagram of an example communication between Host A 31402 and Host B 31404 using a two way stream compression. Host A 31450 has a transmit channel TxA 31450 and a receive change RxA 31460. Host B 31404 has a transmit channel TxB31470 and a receive channel RxB 31480.

In this example, host A initially sends the message A1. To do that it compresses 31408 A1 in to A1` by using the transmit A->B shared dictionary as in the prior art compression (which is initially empty if this is the start of communication between these two devices), and sends 31410 A1` to host B, with the prefix [0] which means that it currently is not using any messages received from Host B to compress this message.

Host B then receives this message (in its receive channel), and decompresses A1` in to A1 using the TxA->RxB shared dictionary which is currently empty, and sends A1 to the requesting application. It then sends A1 to RxB for compression, and then discards the result. In this manner, message A1 gets included in to to the TxB->RxA dictionary.

Host A then sends message A2 to Host B. It therefore compresses 31418 A2 in to A2`, and sends the resulting A2` to host B with the prefix [0] which means that it still has not received any messages from B and thus has not used any messages from B in the shared dictionary when compressing message A2.

At about the same time, Host B sends message B1 to Host A. This happens after Host B has received message A1, but has not yet received message A2. It therefore compresses 31424 B1 in to B1`, and then sends 31426 the message B1` to RxA, with the prefix [A1] which means that it has already seen message A1, and thus Host A should assume that A1 is part of the dictionary when decompressing this message B1.

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 59 of 85

Host A receives message [A1]B1`, so first decompresses 31432 A1` in to A1 so that the RxA channel's shared dictionary includes the message A1, and then decompresses 31434 B1` in to B1 and sends B1 to the requesting application. It then sends B1 to TxA to compress 31436 it in to B1` and discard it, so that B1 is entered in to the TxA's shared dictionary.

Host B then receives message [0]A2`, and therefore RxB decompresses 31428 it with its current TxA->RxB shared dictionary which includes message A1, and sends it to the requesting application. Following the decompression, RxB's shared dictionary now has message A2. The message A2 is now sent to TxB so that it compresses 31430 it, and discards it, so that it is added to the TxB->RxA shared dictionary.

Host A now wishes to send message A3 to Host B, so it compresses 31440 it to A3` and then sends 31442 it as [B1]A3`, since B1 is the last message received from Host B before sending this message.

Host B receives [B1]A3`, so it first decompresses 31444 B1` and discards the result, so that B1 is part of the TxA->RxB shared dictionary on host B, and then decompresses 31446 the message A3` using the new shared dictionary and sends the result to the requesting application. It then sends A3 to TxB for compression 31448 and discards it, so that message A3 is now part of the shared dictionary TxB->RxA.

Figure 318 is a flowchart of the two way stream compression.

In 31802 the unit initializes some parameters: host_last_rec=0 [last msg number this host received], dest_last_rec=0 [last msg number other side received], dest_last_rec_prev=0, msg_counter=0.

Then the unit checks 31804 whether any messages were received from the network (ie. From the other host this unit is communicating with). If so, then the format of the message 31818 is of the type [dest_last_rec]new_msg[n], i.e. the index of the last message the other side has fully received from this host, the new message which we will call new_msg[n]` for the sake of this diagram. For example, a message that we will call [B3]A2` means that the other side (host A) has sent a message saying that the last message it has seen from this host (host B) is B3, and this message that the other host is sending is called A2` meaning that it came from host A, it's the second message, and its compressed (thereby the `).

In 31820 it is checked whether the dest_last_rec (i.e. the index of the last message received from this host) is greater than the dest_last_rec_prev (i.e. the index of the previous known last received message.

If it is, then this host decompresses 31822 all the messages from the last known received message that it sent to the one indicated by dst_last_rec, in order to synchronize the shared dictionary in its Rx channel with the Tx channel of the other side.

It then sets 31824 the dest_last_rec_prev to be the current dest_last_rec, and decompresses 31826 the new_msg` to new_msg using the known prior art techniques, and sends 31828 new_msg to the requesting application (i.e. the application that is responsible for the communication with the other host).

Then in 31830 the new_msg is compressed and discarded, so that the shared dictionary on Tx channel of this host is synchronized with the Rx of the other host, and host_last_rec is set 31832 to n, since that is now the last message received, and control is returned to 31804.

If in 31804 it is found that a message was not received from the network, then in 31806 it is checked whether a message was received from input, i.e. from the application responsible for the communication with the other host. If there was no such message received, then control resumes with 31804. If a message was indeed received from the input, then in 31808 we state that the message is called 'NEW_MSG' for the sake of this algorithm. In 31810 the msg_counter is increased by one, since there is another message to transmit. In 31812 it sets msg(msg_counter) is set to new_msg, and in 31814 msg(msg_counter) is compressed to msg(msg_counter)`. Then this message is sent 31816 to the other host in the following format: [host_last_received]new_msg`(msg_counter).

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 61 of 85

WIRELESS

Figure 400 shows a setup of a network, where a communication device 40002 is in the vicinity of several devices that it can connect to, in this case AP1 40004, AP2 40006 and AP3 40008, where some may be password protected by various schemes such as WEP (see http://en.wikipedia.org/wiki/Wireless_security, http://en.wikipedia.org/wiki/Wi-Fi_Protected_Access, http://en.wikipedia.org/wiki/WEP for several such wireless encryption and locking schemes, which are incorporated in to the document in whole). The communication device 40002 desires to be connected to these devices (for example for connecting through them to the Internet), and so would have benefit in connection to one of these devices. However, if the devices in the vicinity of the communication device 40002 are locked or password protected, then the communication device cannot connect to the them. For such cases, it would be beneficial to have a scheme of guessing the passwords of these devices in such a manner that would be effective.

There are several correlations that can be used to guess the password for the devices:

1. Geographical correlation: Devices that are in a certain country or geographical region are more likely to have certain passwords because of the tendency of the local population to choose those passwords.
2. User correlation: the user of a computing device will typically connect to other devices that have a correlation of passwords amongst them, for example because the user chose the password and will often choose the same password
3. Device correlation: devices typically have a default password from the manufacturer which are often unchanged.

Figure 402 provides a sample implementation of guessing passwords. In 40204 the device tries the most common passwords in that local geography. This database can be collected periodically on the Internet and entered in to the connecting communication device. In 40206 the device tries all the passwords that it used in the past to successfully connect to other devices. It can for example use the most recent passwords first. In 40208 the device uses a pre-loaded list of default passwords that are used by various manufacturers of devices, and tries the default passwords used by the manufacturer of that device that it is connecting to. Typically in the communications handshake the device and/or manufacturer name is provided.

Claim:

1. **A process for automatically connecting to a password protected device who's password is unknown, by which passwords are guessed by a series of one or more of the following schemes in any order: by using passwords that the user uses to connect to other devices, by using passwords that are common to the geography, by using passwords that are common to devices from this same manufacturer.**

Figure 404 is a prior art diagram of how devices such as Clien1 40410 and Client2 40420 connect to devices on the network that require authentication, such as Device1 40404, Device2 40406, etc. These devices such as Device1 can be for example access points that the Client1 40410 and Client2 40420 wish to connect to in order to connect to the Internet, but can be any type of device. It is possible that Client1 40410 and Client2 40420 would be willing to share some of the authentication methods, and thus a system for sharing authentication information is desirable.

Figure 406 is a diagram of a system for sharing authentication information such as the passwords for wireless access points. In the diagram, Client1 40610 has connected to Device1 40604 and marked it as a "PRIVATE" connection meaning that Client1 40610 does not want to share the authentication information with anyone else (for example if this is a WiFi access point that connects to his private information at home). It also connected earlier to Device2 40606 and marked it as an "ALL" connection meaning he's willing to share the authentication information with everyone. It also connected earlier to Device3 40608 and marked it as a "FRIENDS" connection meaning he's willing to share the authentication information with everyone that is defined as his friend on the system. Client2 40620 is defined as a 'friend' of Client1, and thus has the authentication information (though not necessarily in user readable form) required to connect to Device2 40606 (because its defined as "ALL"), and to Device3 40608 (because its defined as "FRIENDS"). 40650 is a central user and authentication database that keeps store of the connections between the users of the system (i.e. who is friends with who), and also of the authentication between clients. The clients update the central database 40650 when they have new authentication information about a device, such as when the authentication information no longer works, or when authentication information is known about a device for which it was previously unknown.

Once this system is deployed in large numbers, the size of the authentication database can be quite large. Thus the update from the central database 40650 to the clients can be done in parts – i.e. to load only the information that the client is most likely to require, for example to limit the size of the database to local geography, and/or by getting all access points in close proximity to all (popular) points of entries in to countries.

Figure 408 is a flowchart of how implementation of such a system may be done. In 40802 a connection is periodically created between the client and the central user/authentication database. In this update, the client gets an update on authentication information to devices that may be of interest to that user, and also updates the central database of any new information it has acquired on authentication methods to devices. The size of the database that is loaded in to the client may be limited as previously mentioned.

In 40804 it is checked whether the client wishes to connect to a device. An example could be because the client wishes to connect to the device which is a WiFi access point in order to connect to the Internet. If it does wish to connect to a device, then it is checked in 40806 whether this device needs authentication. This can be checked through the protocol handshake with the device for example. In 40808 it is checked whether the local database of the client has the authentication information about this device. If it does, then it uses 40810 this information to log on to the device, and tries to log on to

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 63 of 85

the device using this authentication information. If not successful, then the Client updates its local database that this information is no longer useful, and marks the information to be updated to the central server when next connected, and the user is prompted to enter the authentication information in 40812. If in 40808 the database did not include the authentication information for this device, then in 40812 the user is prompted for the authentication information for that device, and the Client authenticates with the device. If that the authentication is successful 40814, then in 40818 the user is prompted for which level of sharing the user wants to apply to this authentication information. Any levels may be defined in an implementation of such a system. For example, the levels may be "all", "friends" (i.e. only share with friends), "family" (i.e. only share with family), "none" (i.e. don't share with anyone). If the user chose to share the password with any number of people, then the Client updates 40820 the central server with the authentication information as well as the information about the device, and the level of sharing with other clients.

Claims:

1. A system of sharing authentication methods between Clients that wish to connect to devices that require authentication, which includes user using Clients, a central server or servers that contain a list of known devices and authentication methods for each and information about social connections between users, clients that wish to connect to devices, and devices that require authentication, where when a user uses a client to connect to a device and enters authentication information for the device, the user is prompted for a sharing level of the information with others, and where that information is provided back to the central server and used by other Clients to connect to these devices
2. The system in claim 1 where the devices are wifi access points

When a client is connected to a communication device it sometimes disconnects and later reconnects. The disconnection (also called a 'cutoff') can occur for various reasons which may be categorized in to two categories: 'explicit user disconnect' reasons – which are the cutoffs due to explicit user request to disconnect from a device or because of poor service from the device, and 'other' reasons such as turning off the client device, disconnecting the client from the network (for example by turning off the wifi switch), etc.

When clients try to connect again to a device (e.g. a wifi access point), they typically have 'favorite' devices, which are devices that the client has already connected to in the past, and the client tries to connect to these devices first.

Figure 410 is a prior art diagram of a client 41002 that is trying to connect to access points that are defined as favorites 42006.

It is possible that the client 41002 disconnected explicitly from one of these devices in the favorites group (for example because the service he was getting from that device was poor), and then when the client software again tries to connect to a device, it would try to connect to that poor service device again, providing again poor service.

It is therefore beneficial to have a system that would distinguish between 'explicit user disconnect' reasons and 'other' reasons within the favorites group, and would first try to connect to the 'other'.

Figure 412 shows such a system, where within the favorites group 41206, it subdivides that group in to two new groups: "explicit user disconnect" group, which is the group of devices which the user chose to disconnect from, and an "other" group which is a group of the favorites (in this case with only one device member) that was disconnected from the client because of 'other' reasons, and this device is the one that the client will try to connect to first.

Figure 414 is a flowchart of how such a system could be implemented. In 41402 it checks if the client has just disconnected from a device. If it has, then in 41404 last_dv is set to be the device that was just disconnected from. In 41406, it is checked whether the disconnection was done proactively by the user, i.e. by the user pressing 'disconnect' on that device., and in 41408 it is checked whether the disconnect was done because of poor performance from the device, such as the device not responding, or responding slowly, etc. If the answers to both questions are no, then last_dv is marked as "other" 41410 and the Client tries to connect to it first, and afterwards tries the other "Other" devices, and lastly the "explicit user disconnect devices". If the answers to both questions are yes, then last_dv is marked as "explicit user disconnect" 41412 and the Client tries to connect to the devices marked as "Other" devices, and lastly to the "explicit user disconnect devices".

Figure 420 is a flowchart of an algorithm that describes how clients of wireless access points can attempt to connect to multiple wireless access points around them in such a method that would on an average case cause the clients to connect to one of those wireless access points quicker.

Image resizing

In computer graphics, image scaling is the process of resizing a digital image. Scaling is a non-trivial process that involves a trade-off between efficiency, smoothness and sharpness. As the size of an image is increased, so the pixels which comprise the image become increasingly visible, making the image appear "soft". Conversely, reducing an image will tend to enhance its smoothness and apparent sharpness. See more about image scaling at http://en.wikipedia.org/wiki/Image_scaling which is incorporated in to this patent as a whole.

Typically, rescaling of images (making them smaller or larger) involves manipulation of one or more pixels within the original image to create a new pixel in the target image.

In many of its applications, this process needs to be efficient to enable it to run in real time. Being able to manipulate less pixels in the original image while achieving similar outcomes will typically increase the efficiency of the process. Thus, a process for scaling images by reducing the amount of pixels that needs to be manipulated that is more efficient is desirable.

Figure 500 shows the prior art method of reducing and increasing image sizes. Block 50010 shows a process of increasing the size of an image. 50016 is a pixel in the resulting image. 50012 is a portion of

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 65 of 85

the pixels in the source image. To create each of the pixels in the resulting images, only one pixel (or two pixels at worst case) in the source image are required to be manipulated to create the larger image.

Block 50020 shows a process of reducing the size of an image by up to 50% (i.e. 0% to 50%). In this process, the new pixel 50024 in the target image is created by some manipulation on 4 pixels 50022 in the source image (by averaging out the four pixels for example).

Block 50030 shows the same process of reducing the size of an image by up to 50%, but in a worst case scenario. The worst case is that 9 pixels 50032 in the source image need to be manipulated to create a pixel 50024 in the target image.

Block 50040 shows the process of reducing the size of an image by over 50% (i.e. 50%+ to 100%), in this case by reducing it 3X of the original image size (i.e. a 66.66% reduction in size). For reducing by 3X, at worst case 16 pixels 50042 in the source image are to be considered to create a pixel 50044 in the target image. More generally, to reduce an image by N X times, (N+1)^2 pixels are to be considered (e.g. to reduce an image by 10 X times, 11^2=121 pixels are to be considered).

A different way of achieving similar results could be the following; if an image needs to be reduced to less than 50% of its size, reduce it in stages, where in each stage the image is reduced by 2X, thus involving 4 pixels in the source to create 1 pixel in the target, then repeat this process until the resulting image needs to be reduced by less than 50% to its final goal, and at that point work according to prior art that involves at worst case 4 pixels.

Figure 504 is a flowchart describing how a more efficient method could be implemented. In 50402 the input for the process is received; the original image (called img[0]), and the scale of the resize (called resize) in % (where resize is is the size of the target image relative to the source image. For example, if the new image is to be 4x smaller, then the resize is 1/4x=25%). In 50404 N is set to 1, and in 50406 it is checked whether the resize is over 50% (i.e. less than 2x downsize is required). If so – then the prior art method is used. If however the resize required is under 50% (i.e. more than 2x downsize is required), then a new temporary image called img(N) is created by downsizing the image(N-1) by 2x (50%) as with prior art, where at most 4 pixels are manipulated for each resulting pixel in the target image. In 50412 the resize variable is doubled, and N=N+1, and flow is sent back to 50406.

HEREDETARY DRAG:

In prior art, when an element of a user interface is dragged or otherwise moved somehow, it is dragged until reaching the outer limits of its parent object.

Figure 510 is a diagram of prior art. In block 51010 it is shown that in prior art systems where object 1 is within a hierarchy with object 2 51014 and is below it in that hierarchy, then when object 1 51012 is dragged towards an edge of object 2 51014, it is stopped at the edges of object 2 51014, as is seen in block 51020, even if object 1 is attempted to be dragged further.

Since in such a case as described the user attempted to further drag the first object past its parent's borders, there is often value in allowing that object to be dragged further. This could be done by 'inheriting' the drag of the lower level object in to the higher level object until that higher level object reaches the boundaries of its parent, and then 'inheriting' the drag to its parent, etc., until the initial object is not dragged any longer.

Figure 512 is a diagram showing this new concept of inherited drag, object 3 51216 is the parent of object 2 51214 which is the parent of object 1 51212. In block 51210 object 1 51212 is dragged towards the edge of object 2 51214, and upon reaching that border is dragged further. The result is shown in block 51220 where this dragging of object1 51212 dragged object 2 51214 towards the edge of object 3 51216.

Figure 514 is a flowchart of how such a system can be implemented. In 51402 an object (object[1]) is requested to move from its current position at (x,y) to a new position marked by (new_x, new_y). note that the position could be marked in any other coordinate system, and is not limited to a two dimensional system.

In 51404 N is set to 1.

In 51406 limit_x and limit_y are set by querying the current object's parent for the limits of where this object could move to within the parent object if moved along the path of the requested move.

In 51408 it is checked whether moving to x,y would move the object beyond its parent's limit (limit_x, limit_y). If it is beyond the range, then in 51412 it is checked whether the parent object can continue to move in that direction within its own parent object. If it cannot, then the final new_x, new_y coordinates are set to the coordinates of where object[N] would reach the limits of its parent object[N+1] if moved from x,y to new_x, new_y. If it can, then N=N+1, and the test is repeated on the parent object. This continues until either the object[N] no longer needs to move outside of its parent object, or until when the parent object can no longer move, and then in 51410 the object is moved along with its parent objects that are participating in this move, to their final location.

Claim:

1. **A system for user interface design wherein exist certain objects that are in a hierarchical structure (i.e. some of these elements are sons or parents of other elements), wherein if one object is dragged past the borders of its parent object, that dragging is carried over (inherited) to its parent object, and so on recursively towards that parent's object**

CACHING

Sameness between URLs

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 67 of 85

In caching systems where caches hold copies of queries and their associated data (e.g. URLs and their corresponding data), there can be situations where multiple URLs in fact contain the same content. In such a case, it would be wasteful to fetch the content from the network server that holds the content if the content in fact is stored locally in a cache. Therefore, it is beneficial to create a system that given a URL and a second URL which is 'suspected' of holding the same data, would be able to determine if both URLs are in fact pointing to the same data and whether the cached data is still valid.

Figure 600 is a diagram of prior art showing this need – communication device 604 needs to fetch the contents of a URL for which it has a copy in its local cache 608. The URL's data is also located both on the data server 606 and on a Peer communication device 612 in its cache 610. Since the local copy of the data has expired (its validity has expired), it needs to check whether the data in its cache is the same as the data in the data server 606 or in the Peer communication device 612. However, there is no prior art method of doing this, and thus the communication device 604 will re-fetch the data from either of these sources in full.

Figure 602 is a flowchart of a system to enable checking whether the cached data is the same as the data in the other sources. This is done by determining the number of bytes to load from the external source in order to gain a reasonable probability that the information stored locally is identical to the valid data. For example, it can be determined that fetching a data range that is 5% in its size of the full data that is identical to the data available for that URL in the cache provides reasonable enough probability that the cached data is valid.

In 60002, the module receives from its caller a URL that the communication device needs to fetch, as well as its entry within the communication device's cache which is expired (if its not expired, then the communication device would simply use this entry as the result of the URL query).

In 6004 the module receives from its caller the place where a valid copy of the data for this URL resides. This can be on a data server, such as a web server, from where this data copy originated, or could be a Peer on the network (for example in an acceleration system) that has a valid copy cached.

60014 checks whether the cached URL's validity has expired. If not, then in 60016 the cached URL is returned. If the cached URL has expired, then in 60012 a range request for X bytes is sent to the external source (could be 16kb for example, or could be a random range of 5% of the size (in bytes) of the cached URL's size.

In 60018, the headers of the returned data are analyzed. If the ETAG and other file identifiers are the same, then the local cache's validity is updated to that of the headers, and 'use local data' is returned in 60016.

If the headers do not match the headers of the cached copy, then in 60020 the range that was returned is compared to the content in the cache, and if identical, 'use local data' is returned in 60016. If the data segment is not identical, then 'different data' is returned to the caller in 60024, and the caller will most likely fetch that data again from the external source.

A modification could be in 60012 to only fetch a few bytes to receive the headers, and then only if they do not match the headers in the cache to ask for a range.

In caching systems, a local cache may store a copy of data to be retrieved (such as a URL and its data), and its corresponding validity, which is until when can this data be used. After the period of the validity this data has to be re fetched from its source server.

Thus, when a certain data is requested, the communication device will query its cache, and at that critical time may find that the data in the cache is invalid, and will need to spend time to fetch this data.

It would be beneficial to have a system that keeps important information in the cache valid without degrading the performance of the system.

Figure 610 is a flowchart for how this can be done, by validating invalid (or soon to be invalid) cache entries while the communication device is idle.

61002 checks if the communication device's resources are idle (or idle enough) to update entries in its cache without degrading system performance (i.e. checks that CPU is below a certain threshold for example of 30%, that the disk is not in use, that there is excess bandwidth that can be used, etc.).

If the communication device has idle resources, then in 61006 the cache entries that are flagged as 'important' to the system (could be all cache entries for example) are scanned and sorted by time to invalidation, where at the top of the list are the invalid entries, and below them the soon to be invalid entries, and so on.

In 61010 the top item in the list is chosen, and in 61012 a query is sent to the data server to revalidate the entry. This can be done in several ways, for example:

1. To send a request for the full entry and replace the cache entry (this is prior art)
2. To send a request for only the first byte and to decide based on the headers whether this cache entry is still valid – ie. If the ETAG, size and other parameters in the header are the same, to revalidate the cache entry to the validity found in the header
3. To send a request for a segment of a fixed size (e.g. 5% of the file size) of a random location, and if the same as in the cache to use the server's validity setting

In 61014 the result is stored in the cache entry and the entry is removed from the list of revalidation. In 61016 the module checks if there are still enough free system resources to continue the revalidation. If there are then the module continues with 61010, and if not continues to wait in 61002 for idle resources.

In Patent #8135912 (SYSTEM AND METHOD FOR INCREASING CACHE SIZE), a concept of writing cache data to the free portion of the memory is introduced, where this information is transparent to the operating system, and thus more cache size is available without degrading the amount of memory that is available for the user to use.

In such a prior art system, the file system writes the OS data (in 62002), after which the cache data is written (in 62004), after which the completely free space is at (in 62006). The problem with this approach to memory management is that when the file system adds data to the OS data 62002, it overwrites the cache data. In such a system where totally free space 62006 is still available, it would be beneficial to have a system where such further OS writes would not overwrite the cached data while free space is available.

Figure 622 is a diagram of such a system. The OS data is stored at 62012, whereas the cache data 62016 is stored on the other side of the device, starting at the furthest position from the start of write of the OS data 62012, so that the OS data only over writes the cache data when the totally free space 62014 is completely used up.

It is also desirable to have a system that cleans up the cache data, so that data that no longer should be cached is removed, to maintain the free space and avoid cache overwrites.

Figure 622 is a flowchart of a system for writing cached data in this modified method, as well as for cleaning up the cache periodically to allow for free space in the system and thus less data overwrites.

In 62202 the cache_pointer is set to the point on the storage device which is furthest from the starting point of where the OS data is written to when the storage device is empty.

In 62204 it is checked whether cache data (cache_data) needs to be written to the storage device. If not, then in 62206 it is checked whether system resources are idle enough to warrant a cache cleanup to be performed.

If there is more cache to write, then in 62208 the cache_pointer is moved back (i.e. towards the OS data) by the size of cache_data, and then in 62210 the cache_data is written to the storage device as the referenced patent instructs how to do (i.e. without notifying the operating system so that this space is continued to be viewed as empty by the OS). This moving back of the pointer is novel, since it creates a situation where the data is written in a 'forward' direction (i.e. in the same direction which the OS data is written at), which is typically the faster writing direction for storage devices, as they are optimized for writing OS data.

When the cleanup is performed, then in 62214 it is checked whether the free space is close to running out (e.g. is it under x% of the total available storage size, where x can be 10%, or under y Bytes free where Y is 1GB for example).

If this threshold has been reached, then in 62216 the least relevant cache is searched for to be removed. The criteria for less relevant could be in that it has expired, or that it is accessed the least, or any of other prior art cache purge methods. This cache item is removed.

A cache system such as that described in Patent #8135912 (SYSTEM AND METHOD FOR INCREASING CACHE SIZE), creates a very large cache size (cache_size) but at the expense of the reliability of reading the cache back after writing it, where the probability for a cache miss (i.e. to try to read back the cached element and to fail) at a certain point in time would be P, where P<1.

In some cases, the cache_size is significantly bigger than required for the system operation. In such a case, Figure 630 is a diagram of how P can be increased (i.e. reliability can be increased) at the cost of the cache_size.

Figure 630 offers to create two zones of writing the cache data in the storage device, where each element in the cache is written once to each zone. This way, the cache_size is reduced by half (since each element is written twice), and P would be reduced to P^2, since the probability not to read back P would be to miss it in both of the zones. Thus, while the size of the cache is reduced linearly (cut in half), the probability of getting a cache miss is reduced exponentially (P^2).

Figure 632 is a flowchart for how this could be implemented.

In 63202 a cache read or cache write command is received by the module. 63204 checks if it is a read command or a write command.

If it is a read command, then in 63206 the module looks up in the cache index to find the two locations to which the cache was written, and in 63208 attempts to read the cache entry from the first location. 63210 checks if the cache entry was found in that read. If it was, then return 63212 the data that was read. If it was not found, then in 63220 read the cache entry from the second location and return it if found 63212 or return data_not_found if not found in the second location as well.

In the case of a write command, two different free locations are identified in 63214 and the cache data is written to those two locations 63216. The cache index is updated with these two locations so that the cache entry can be found in future writes.

Figure 634 shows that in a similar manner, the amount of times that a cache element is written to the storage device can be increased from 2 as shown in figures 630 and 632, to any number N, where in such a case the available cache size is reduced from (cache_size) to (cache_size/N), and the probability for a cache miss is reduced from (P) to (P^N).

NDCACHE - Non deterministic volatile memory cache

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 71 of 85

In a computing device, the random access memory (RAM) is a limited resource. When the operating system uses the RAM it typically increases the speed of the application. However, excessive use of the RAM for an application limits the use of the RAM by other applications and thus limits their speed. One such use of the RAM is for caching information in order to speed up the speed of the program's operation. Such cached data may be data retrieved from the network, or be the result of a complicated operation, etc. Operating systems make use of the RAM for caching purposes, and typically leave some of the RAM free to be used later. If this free RAM memory could be used to store additional cache without significantly impacting system performance in other ways, that would be beneficial.

Figure 640 is a diagram of the state of the art implementation of address space mapping. A physical address 64004 is a memory address that is represented in the form of a binary number on the address bus circuitry in order to enable the data bus to access a particular storage cell of main memory, or a register of memory mapped i/o device.

In a computer with virtual memory 64002, the term physical address is used mostly to differentiate from a virtual address. in particular, in computers utilizing memory management unit (MMU) to translate memory addresses, the virtual and physical addresses refer to an address before and after MMU translation respectively.

Figure 642 is a diagram of a prior art MMU and TLB, in which the CPU 64002 requires a translation of a logical address in to a physical address 64008 in order to read or write to it. For the translation it sends the logical address to the MMU 64004, which uses a cache called a TLB 64006 to map the virtual address to a physical address 64008. Modern MMUs 64004 typically divide the virtual address space into pages, each having a size which is a power of 2, usually a few kilobytes, but they may be much larger. The bottom n bits of the address (the offset within a page) are left unchanged. The upper address bits are the (virtual) page number. The MMU 64004 normally translates virtual page numbers to physical page numbers via an associative cache called a translation lookaside buffer (TLB) 64006. when the TLB 64006 lacks a translation, a slower mechanism involving hardware-specific data structures or software assistance is used. The data found in such data structures are typically called page table entries (PTEs), and the data structure itself is typically called a page table. The physical page number is combined with the page offset to give the complete physical address.

Sometimes, a TLB 64006 entry or PTE prohibits access to a virtual page, perhaps because no physical random access memory has been allocated to that virtual page. in this case the MMU 64004 signals a page fault to the CPU 64002. The operating system (OS) then handles the situation, perhaps by trying to find a spare frame of RAM and set up a new PTE to map it to the requested virtual address. if no RAM is free, it may be necessary to choose an existing page

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 72 of 85

(known as a victim), using some replacement algorithm or 'eviction algorithm', and save it to another form of storage (e.g. disk) - this is called "paging".

Figure 644 is a diagram of prior art on how an MMU works. The TLB 64402 receives a virtual address from the CPU. If it finds the associated physical memory within the TLB 64402, then it returns the Physical address associated with this virtual memory, to the CPU. If it does not find the address in the TLB 64402, then it looks it up in the Page Table 64404. If there is an association there, then the physical address associated with the virtual address is returned. If there is no such association, then the Page Table Exception Handler 64406 is activated. It uses a database and a set of drivers to figure out how to map the virtual memory to physical memory (often by allocating new memory, loading information on to that memory from disk, and mapping it to a virtual memory). The data is then loaded in to the physical page in 64408, and the physical address is returned to the CPU.

Figure 648 is a diagram of prior art on how the MMU's page table exception handler works. When a page table exception occurs, the page table exception handler is invoked, as per diagram 642. The page table exception handler 64802 first identifies 64804 the driver responsible for this data segment. This is typically stored in the page table. For example, the driver for a specific virtual memory segment may determine that this part of the virtual memory should be mapped to real physical memory, or it may determine that it is mapped to an IO device such as a camera and the contents should be read from the camera's sensors. The driver assigned then determines whether a new physical memory segment needs to be assigned to this virtual address space or not. If more physical memory is required, then in 64808 the OS determines whether there is such physical memory available to the system. If there isn't free physical memory, then in 64808 the OS determines which physical page to 'purge' out of memory based on any number of 'eviction algorithms' (e.g. least recently used is discarded, etc.). The driver assigned to this memory space determines in which manner this information is purged – i.e. is it simply discarded, or saved to disk, etc. The MMU then evicts 64810 the physical page by swapping it to disk, erasing, or whatever method the driver associated with it determined.

If in 64806 it is determined that there is enough free physical memory to create the new page, then the driver determines 64814 the method by which to load the data in to the physical memory or to otherwise map the virtual address to an IO device, etc., and then loads 64816 the data in to the physical memory or maps it as required. The virtual memory map to the physical address is then sent to the TLB 64818, and the request is resolved for the CPU.

The prior art memory management can be described as deterministic, because information that is stored in the physical memory by the application is always retrievable, even in case that physical memory is purged, since the driver for that memory segment that is purged typically

saves the data before discarding it. There is a 'cost' to this determinism, in that there is a load and purge time to the physical memory, since when purging this physical memory it needs to be written to a different medium, and then loaded back from the medium before it can be read. In cases where an application wishes to store cache data in the volatile memory in order to save time in re-calculating an algorithm, or in order to not load it again from the web, or in other cases, it does not need the determinism offered by the state of the art memory management systems, but it does need high access speeds, since if the speeds are not high, then it may be more beneficial to the system to re-calculate the algorithm or to re-fetch the information from the web, and not to do a page fault which is costly in time. In addition, the more physical memory available for the cache, the faster that the applications can run. However, the more physical memory allocated to cache of one application, the less is available for other uses, and thus the system performance may be degraded.

Therefore it is beneficial to design a system that can gain from higher speeds associated with non-determinism in the memory management, and that can use the maximum possible physical memory without deteriorating the performance of other applications.

Figure 650 is a flowchart of how a non deterministic physical memory caching system may be implemented. In 65002 an application requests from the operating system a non deterministic physical memory cache (NDCACHE). The OS allocates 65004 the physical memory for the cache, and maps it to virtual memory, and marks these pages with an 'NDCACHE' flag. If the eviction algorithm of the OS decides to swap out 65006 a physical page which is marked with an 'NDCACHE' flag, then that page is discarded 65010 without being stored back to non volatile memory – it is simply removed from the page table, and will be allocated to a new page and the new data will overwrite the old. If the OS is running out 65008 of physical memory (this may be determined via a variety of ways which will be described later on in the document), then the OS discards one or more NDCACHE pages from memory to clear up memory for the OS. (for example, it may clear up four pages, check if that is enough for the OS, and clear more if needed). When the application wishes to read 65014 a virtual memory that was requested as an NDCACHE memory, and is now a regular memory (since the NDCACHE memory was swapped out and thus discarded), it receives back a 'DOES NOT EXIST' message to indicate that this memory was lost and thus the cache data requested no longer exists. During a read or write transaction to or from an NDCACHE page 65016, that page is locked so that the eviction algorithm does not purge it during that action.

Figure 654 is a suggested application programming interface (API) for an NDCACHE implementation. This suggested API is similar to the file system API (http://en.wikipedia.org/wiki/File_system_API) in POSIX (http://en.wikipedia.org/wiki/POSIX).

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 74 of 85

To start a new NDCACHE type cache, an application first will call the function NDCACHE_OPEN 6504 function, which returns a handle to the NDCACHE (FILEDS) that is referenced by NAME. The NAME that is passed to the function is a unique name for that cache that is used between different processes accessing this page, similar to how a filename is used in the file system POSIX API. If the NDCACHE specified by NAME exists, then the function simply returns a handle to the existing cache. If it does not exist then NULL is returned by the function. However, if the CREATE_FLAG passed to the function is turned on, then upon opening an NDCACHE specified by NAME that does not yet exist such an NDCACHE is created and the function returns the handle to that NDCACHE. If such a cache already exists and the TRUNCATE_FLAG is set, then that cache is erased, and a handle to this empty cache is returned by the function.

NDCACHE_READ and NDCACHE_WRITE 65404 are the functions used to read from the NDCACHE and write to the NDCACHE (respectively) that is referenced by the FILEDS which was initially received from the NDCACHE_OPEN function. The BUFFER is an allocated memory that the application can use to read the information to (in case of NDCACHE_READ) or to write from (in case of NDCACHE_WRITE). The OFFSET is the offset (in bytes for example) within the NDCACHE referenced, to start the read or to start the write from. SIZE is the number of bytes to read or to write.

NDCACHE_CLOSE 65406 is called to close the handle to the NDCACHE referenced by FILEDS. After all handles to an NDCACHE are closed, the OS may decide to swap it out based on its eviction algorithms.

NDCACHE_UNLINK 6508 is called to remove the NDCACHE entry referenced by FILEDS from the operating system's page table, thus effectively deleting this NDCACHE.

A secondary implementation of an NDCACHE may use simple file system mounting. This implementation is simple in that it does not require kernel mode modifications, or some minimal modificaitons to the TMPFS file storage facility that is available on many unix-like operating systems (http://en.wikipedia.org/wiki/Tmpfs).

Background:

A file system is a means to organize data by providing procedures to store, retrieve and update data, as well as manage the available space on the device(s) which contain it. A file system is tuned to the specific characteristics of the device.

There is usually a tight coupling between the operating system and the file system. Some filesystems provide mechanisms to control access to the data and metadata. Ensuring reliability is a major responsibility of a filesystem. Some filesystems provide a means for multiple programs to update data in the same file at nearly the same time.

File systems are used on data storage devices such as hard disk drives, floppy disks, optical discs, memory storage devices, remote servers, etc.

File systems may provide access to data on a file server by acting as clients for a network protocol (e.g. NFS or SMB clients), or they may be virtual and exist only as an access method for virtual data (e.g. procfs). This is distinguished from a directory service and registry.

A File system is implemented in a file system driver which implements a standard file system API for using it, and is coupled to a storage device or other storage strategy (like mapping to existing memory).

Figure 662 shows the prior art method of mounting a file system. A computer 66202 has a file system directory structure 66204 which in which various directories represent space on various storage devices. The block device 66208 is such a storage device on which data may be stored. The file system driver 66206 links between the block device 66208 and a certain directory in the file system 66204. the file system driver is configured to be called by the OS on a specific point in the os directory structure (in this example in "/tmp"). Every file call to that directory initiates a call to the File System Driver 66206, that may use a memory device such as the block device 66208 to store & retrieve information. This linking of a storage device to a directory is called 'mounting a file system'.

Figure 664 is a prior art diagram of the TMPFS file system. TMPFS is a common name for a temporary file storage facility on many Unix-like operating systems. It is intended to appear as a mounted file system, but stored in volatile memory instead of a persistent storage device. A similar construction is a RAM disk, which appears as a virtual disk drive and hosts a disk file system. The computer 66402 has a directory structure 66404 in which the TMP directory is mapped to the TMPFS file system driver 66406, such that when file system calls are made on files within the TMP directory, they are performed on data that is stored in a volatile memory 66408.

Figure 666 is a flowchart of a possible implementation of an NDCACHE by using a TMPFS file system. In 66602 a TMPFS is created and mounted, with an initial size of X bytes, where X can be 1GB for example, to be used for the NDCACHE.

In 66604, the system checks whether the operating system would benefit from having additional volatile memory. This can be done in various methods known to those in the art. For example, this can be done by checking every Y ms (for example every 20ms) if less than a certain amount of free memory exists in the system (could be less than 200MB for example), or if above a certain amount of memory swaps occurred recently (could be more than 2 swaps per

second for example), or if the kernel cache space is reduced to under a certain size (could be under 2GB for example). This can also be done by listening to 'stress' operating system calls such as "low memory" type calls – if these are called then the operating system probably would benefit from having additional volatile memory. This could also be checked by attaching a callback on the swapping out of memory of the TMPFS file and deleting it instead of swapping it out.

If any of the above events in 66606 occurred, then the OS would benefit from having more volatile memory. This is checked in 66608. If the OS would not benefit from more memory, then continue to 66604. If it would, then in 66610 the NDCACHE is reduced by deleting a number of elements from it, so that its size is reduced by Z bytes (for example by 200MB). 66612 shows various strategies by which elements may be reduced from the NDCACHE: delete the least recently used (LRU), the least frequently used (LFU), the largest elements by bytes, or the smallest elements by size. This TMPFS size reduction thus frees space for the operating system. This system described in Fig. 666 thus describes an implementation of an NDCACHE, since the application gains a physical memory cache, that is swapped out (partially or completely) as the operating system requires more physical memory for its operation.

CLAIMS:

1. A method of increasing the size of the memory available to applications by which an application may request memory which is specially marked, where the application is able to read and write to this physical memory, and where if the operating system needs more memory then this specially marked memory is discarded
2. The method of claim 1, where if the specially marked memory space is swapped by the operating system, then it is simply discarded
3. The method of claim 1, where if the operating system would benefit by having more memory available to it, it reduces the size of the memory allocated to the specially marked cache
4. The method of claim 1, where the memory is physical memory only
5. The method of claim 1, where it is implemented by a TMPFS system that allocates physical memory to this special cache, where if the operating system would benefit by having more memory available to it, it reduces the size of the TMPFS allocated to this specially marked cache

There is a downside to the second implementation of the NDCACHE -- the calls to TMPFS are from user mode to kernel mode, and thus take thousands of CPU cycles to complete. It is thus

desirable to avoid calls between user mode and kernel mode if possible, to improve performance of the system to get to several cycles per NDCACHE call.

A system with higher performance can be achieved by implementing the NDCACHE as a user mode module that communicates with an NDCACHE kernel module, as described in Figure 672. 67202 is the user mode space of the computing device, and 67208 is the kernel space. The application 67204 is running in the user mode 67202 and using an API (NDCACHE calls) that is implemented in an NDCACHE user mode implementation 67206. It uses system calls to communicate with the NDCACHE kernel mode implementation 67210.

Figure 674 is a description of the API for this third implementation method of the NDCACHE.

67402 is pseudo code that describes how an NDCACHE would be allocated and written to. First, a FILEDS handle is created by opening a file that is handled by the NDCACHE_DRIVER which is the kernel module for NDCACHE events, along with the NDCACHE_NAME which is the specific name of the NDCACHE that is being accessed. Then, a pointer directly to the NDCACHE memory is received by doing an NDCACHE_LOOKUP on the NDCACHE_NAME. This lookup returns a pointer to the memory of the specific NDCACHE, and if such a cache does not exist, P is assigned NULL. If indeed no such NDCACHE exists, then it needs to be created. In such a case, the pointer P is assigned a memory map of a certain size (in this specific implementation it is 1MB, but this could be other sizes), where this memory is declared as shared memory so that it may be used by multiple processes.

Then the first memory position in P (P[0]) is increased (so that if this was a new assignment of memory, then P is now 1), to indicate that there is content in this NDCACHE. Then the program continues by writing the data to P, starting at P[1]. If the cache exists (i.e. was not allocated by this call), then do the NDCACHE_WRITE() function.

Figure 675 shows how such an allocation would look like – the first byte of the allocation is the lock flag (P[0] in the above code), which indicates that there is content within this NDCACHE of 1MB that was allocated, and the 1MB is comprised of a series of 4KB allocations.

Figure 676 is an implementation of the NDCACHE_READ() and NDCACHE_WRITE(). First it is checked whether the cache still exists, by checking the first byte of the NDCACHE range. If it is zero, then that NDCACHE was swapped out, and thus the program returns to the application MEM_NOT_FOUND. Then the program increments the first byte of the NDCACHE range. If it equals '1', then it was zero when incremented – i.e. the NDCACHE range was swapped out and cannot be used. The program then returns the first byte to zero, and returns MEM_NOT_FOUND to the application. Now the NDCACHE range is locked (because the first

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 78 of 85

byte is non-zero), and the range is read in to buffer or written from the buffer to the range, as the case may be for NDCACHE_READ or NDCACHE_WRITE respectively.

After finishing the read or write, the program then decrements the lock flag to release the lock on the range by indicating it has completed the read/write and returns.

Figure 678 is the implementation of the eviction algorithm of the NDCACHE kernel module (the NDCACHE_DRIVER). In prior art operating systems, the eviction algorithm is called when the OS needs more physical memory – it then calls the eviction calls of the various drivers handling memory for them to clear out selected portions of the memory. Some of these drivers are programmed to copy segments of the memory in to other storage devices (such as a hard drive) and then to clear the physical memory for other uses, while other drivers may choose other forms of action.

The eviction algorithm for the NDCACHE_DRIVER is described in block 67802. Upon being called, it removes all associated memory ranges who's LOCK_FLAG is 0 (which means that the memory range is not in use). If more memory needs to be freed, then it can also remove the ranges who's LOCK_FLAG is 1, since they are in use but not locked. It does not remove memory ranges who's LOCK_FLAG is greater than 1, since they are currently locked by a read or write operation. As an example, there may be several NDCACHES in use, each of them of various sizes. When the eviction algorithm is called, it removes those NDCACHES who's lock flag is 0 or 1, but not greater than one. Further implementations of this may choose to not remove all the NDCACHEs possible, but only enough NDCACHEs so that the OS has enough free memory (e.g. 50MB) to continue normal operation.

Block 67804 describes how the swapping out of memory ranges by an NDCACHE_DRIVER is done: Once the eviction algorithm of the NDCACHE_DRIVER decided to free a range, it maps the virtual memory pointers to an "all zero" range in 'read only' mode (for example by mapping to devzero. This quickly sets the memory and the LOCK_FLAG to zero, thus freeing the physical memory for the OS to use, and marking it as empty towards the applications using the NDCACHE.

Figure 680 is a diagram that shows the system for the fourth method of implementation for the NDCACHE, which allows for parts of the NDCACHE to be removed while keeping other parts in the memory. For example, where not all the memory ranges handled by the NDCACHE driver are attempted to be freed, but only a certain amount that will enable the OS to have more free space to operate (e.g. freeing ranges until 200MB are free).

The problem with previous implementation methods: is that an NDCACHE element in the previous implementations is assigned in one block. The problem is that this whole block is swapped out even if only a certain part needs to be swapped out by the OS (all or nothing type

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 79 of 85

of approach). It is desirable to have an implementation where only portions of the allocation may be swapped out (for example, the NDCACHE element could be a whole file, and it may be useful if parts of it are swapped out).

68002 are a group of pages in the physical memory of a computer (page #1, Page #2, … Page #N), in this example each of the pages is a range of 4KB.

68004 are a group of flags in an area of the physical memory allocated to be the management area of the NDCACHE, where there is one such flag for every page in the memory that is allocated to the NDCACHE. Each flag in the management area in 68004 is actually a 'lock flag' from the previous implementation of the NDCACHE and is used in the same way, but is done per page, so that each lock flag correlates with one page in the memory. Thus, the NDCACHE may allocate a large file from Page #1 to Page #N, and when the OS needs more free memory then the NDCACHE's eviction algorithm may choose to free a certain memory size for the OS (e.g. 4MB), and only remove the first 1,000 pages from this NDCACHE, instead of removing the whole cache.

Claims:

1. **An NDCACHE system as described in previous claims, where each NDCACHE is mapped to a multitude of physical memory pages, and where there is one lock_flag associated with each such physical memory page, so that when the operating system of the computing device needs more physical memory, it can release only some of the physical pages associated with this NDCACHE.**

Figure 682 is a flowchart for implementing the fourth implementation of an NDCACHE. When an NDCACHE is requested by an application, the application calls the initialization function and specifies the size of the requested NDCACHE. In 68202, physical memory is allocated for the NDCACHE as was requested by the calling application (e.g. 20MB). The prior art OS responds by allocating such a memory segment (if available), where this memory segment is provided as series of blocks of physical pages, of a certain size as determined by the OS (e.g. 4MB per page).

In 68204, a management area is allocated, where in this area there is one LOCK_FLAG for every physical page that is allocated in 68202. Thus this memory size is equal to [MEM ALLOCATED]/[SIZEOF(MEMORY_PAGE)]*[SIZEOF(INT)]. For example, on an OS where a page size is 4KB, the memory allocated for the NDCACHE is 20MB, and an integer is 4 Bytes, the size for allocating a management area is = 20MB/4KB*4B = 5k * 4B = 20kB (for storing 5,000 flags of 4 bytes each) .

The normal operation (read/write) of the NDCACHE in the fourth implementation is similar to the earlier implementations, but where the memory for the NDCACHE is the series of memory pages allocated to the NDCACHE, and the LOCK_FLAG is not for the whole memory range allocated to the NDCACHE, but for each of the pages in that memory range, and when the eviction algorithm needs to free memory for the OS, it can evict a portion of the memory ranges, freeing up the memory range and its corresponding LOCK_FLAG from the management area.

Figure 690 describes an improvement in the fourth method of implementation of an NDCACHE. The following are three problems with the fourth method of implementation that may be improved:

1. When an NDCACHE that is smaller than the OS's memory page size (typically 4kB) is requested, a full page (e.g. 4kB) is allocated and thus there is waste in allocating more than is needed.

2. When an NDCACHE that is larger than the page size is requested, the allocation needs to be done in multiple parts (once for each page), thus degrading performance and degrading the ease of use of the API.

3. The fourth implementation does not conform to the 'malloc'/'free' paradigm, which makes it more difficult to integrate in to existing and new solutions

In this improvement, when a new NDCACHE is allocated, when allocating the pages in memory 69002 for the NDCACHE, a secondary management area 69004 is allocated at the first page, such that it includes a pointer to the first LOCK_FLAG in the management area 69006, and a counter of how many pages are allocated to this NDCACHE ("N") is maintained. Thus, when allocating multiple pages for an NDCACHE, the secondary management area 69004 includes all the information needed to use all pages and lock flags.

Figure 692 is a flowchart of the initialization process for the improvement on the fourth method. In 69202 when the NDCACHE is requested by the application, a memory range (P) is allocated for the NDCACHE. Its size is the size of the memory requested + sizeof(T) where T is the area required for the secondary management area.

Then in 69204 space is allocated for the management area to hold the LOCK_FLAGs for each of the memory pages required by the new NDCACHE. Then, a secondary management area (T) is defined in 69206, and into it is inserted a pointer (T->PTR) to the management area and an integer (T->N) that represents the number of pages in this NDCACHE's memory range. In 69208 the pointer P is changed such that P=P+SIZEOF(T). This is so that P now points to the start of the main memory range (right after the secondary management area). Lastly, in 69210 the

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 81 of 85

program returns the pointer (P+SIZEOF(T)) to the calling application, which is a pointer to the place in the memory where the NDCACHE memory itself starts (right after the secondary management section).

Note that the allocations for these memory ranges can use any of the existing malloc/free implementation algorithms to allocate memory from the main NDCACHE pool to this allocation. the malloc and free can be done over mmap with the implementation of NDCACHE shown in the fourth implementation, which makes this a non-deterministic cache.

Also note that if the allocation is less than one page size, T->N could still span more than one page, if for example this page already contains part of a previous allocation and this this new allocation causes the allocation to overrun to the next page.

Figure 696 is a flowchart of the actions of reading from or writing to the NDCACHE for the improvement on the fourth method of implementation. In 69602 the read/write is described. The read/write is done in the same way as in the fourth implementation with two exceptions:

1. Lock(P)/unlock(P):
    a. First evaluates that T->N is not zero. If it is zero, then the NDCACHE was swapped and needs to return mem_not_found
    b. If T->N is not zero, lock all pages that this NDCACHE element spans by locking the N LOCK_FLAGS starting at the T->PTR LOCK_FLAG.
2. When reading/writing from/to the NDCACHE, not limited to one page of memory (can use the complete allocated size)

In 69606 the deleting of the allocation - NDCACHE_CLOSE(P) – is described. The NDCACHE is freed by calling the OS free() function and providing to it the pointer of (P-SIZEOF(T)) where P is the pointer to the main memory range, and T is the secondary management area, so that the memory freed is both the secondary management area as well as the main memory area.

A further note about NDCACHE: There is an advantage to being able to use named objects for NDCACHE – i.e. for the NDCACHE objects to be named so that they can be used by multiple processes, and also lets the virtual memory related to these named objects to be freed on swap.

This can be implemented over the systems described previously by adding a hash table where in the open/create of the object, this name is looked up in the hash table, and if found provides back a pointer to an existing object.

OTHER

There are applications that may benefit from understanding when the computer's resources are idle. For example, a screensaver monitors the mouse and keyboard movements for idleness, and after a preset amount of idle time it activates its display program. Other cases are for a web site to monitor in-activity of keyboard or mouse input, and to log out as a consequence.

Figure 700 is a flowchart of an idle monitor that uses new inputs to define idleness of a computing device or its operator.

In 70002, an application wishes to be notified of when idleness has occurred, and so notifies the idle monitor program, provides a callback function (CB) - a function to call when the idleness occurs - and defines the type of idleness on which to call this function (PARAMS) and the duration of time they should be idle for the CB to be called.

The program then scans 70004 'idleness' resources and tracks for how long each has been idle. These resources include the following:

- Bits being sent or received to/from communication device – i.e. idleness of communication
- Storage device read or write – i.e. idleness of the IO with storage devices
- Temperature changes in any of the device's temperature gauges – i.e. idleness of the environment. Non-idleness of the environment could signal that there is movement in the environment of this computing device, or that the temperature around is is shifting
- CPU busy over certain threshold (could be 5% for example) – i.e. idleness to a degree of the computing resources
- Camera senses motion – i.e. idleness of the physical surrounding of the computing device (movement of persons for example can trigger this to not be idle)
- Light sensor senses changes in light intensity or structure - i.e. idleness of the physical surrounding of the computing device (movement of persons for example can trigger this to not be idle)
- Accelerometer that senses movement or orientation change
- HD accelerometer that senses movement
- GPS sensor that senses movement

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 83 of 85

If any of the above sensors is idle 70006, then check 70008 whether all sensors included in PARAMS are idle and have been idle for the amount of time described in T. If they've all been idle for this time, call the CB.

CLAIMS:

1. **In a system monitoring the idleness of a computing device, the following sensors may be included in a list of devices to be monitored for idleness: Bits being sent or received to/from communication device , Storage device read or write, Temperature changes in any of the device's temperature gauges, CPU busy over certain threshold, Camera senses motion, Light sensor senses changes in light intensity or structure, Accelerometer that senses movement or orientation change, HD accelerometer that senses movement, GPS sensor that senses movement in location or hight**

REDUCING STORAGE READ TIMES

In storage devices with moving media, physical seek times are typically long relative to the read or write times because of the read/write head movement, and thus should be avoided where possible

Also in storage devices, writing to the storage device is typically less urgent as reading from it, since the writing is typically to archive existing information, whereas reading data is used for actionable results (such as displaying data on the screen) and thus may be blocking.

In prior art systems, reading and writing occur randomly, and thus when the typically higher priority reading is occurring, it could possibly be after a write has occurred, and thus the reading head is typically not close to the place where the information should be read from.

Delaying the writing of data until storage device read/write is idle can speed up the read time, since there's then no need to wait for write to end before reading and also eliminates many seek cycles.

Figure 710 is a diagram of a system to reduce the read times from a storage device 71008. It includes the OS 71002, which calls the storage read time reduction module 71004 when accessing the disk. This module uses the storage device idle monitor 71006 to determine when the storage device is idle. When the storage read time reduction module 71004 wishes to write to disk it first ensures that the disk has been idle for a pre-determined amount of time (eg. 30ms) before writing to the storage device. If the storage device is busy, it increases the chance that while the OS is writing data to the disk, a disk read will occur, which will be slowed down by the read time and seek time back to the place to read from. Thus, with the system described

NetNut Ltd. v. Bright Data Ltd.
IPR2021-00465, EX. 2022
Page 84 of 85

by the diagram in figure 710, such conflicts (of having a read while a write is in action) are reduced. Thus when the storage device idle monitor reports non-idleness, then the data to be written to the disk is written to the write queue 71010 instead of to the disk.

This concept may be broadened by looking not only at the idleness of the storage device 71008 by the storage device idle monitor 71006, but by incorporating an idleness monitor that checks for a much broader set of idleness parameters such as keyboard inputs, mouse inputs, network communication, etc., since any kind of non-idleness on the computing device typically correlates to reads from the storage device, and thus storage writes should be avoided during such periods of non-idleness.

CLAIMS:

1. **In a computing system comprised of a storage device and a processor, a method for reducing the average time to read data from such storage device, by which storage writes are only performed when the storage device has been idle for a certain amount of time (eg. 30ms)**
2. **The method in claim 1, where other parameters are checked for idleness, such as the keyboard input, the mouse input, the network communication, etc.**

Figure 712 is a flowchart of an implementation of a storage read time reduction module SRTRM. If the SRTRM received a write command 71202 then it checks 71204 whether the write queue has enough free space for queuing this request. If it does, then the request is queued 71206 until the storage device is idle at which time it will be written to it. If the write queue does not have enough space to queue the new write data, then the program writes 71205 the new request data to the storage device.

If the SRTRM has not received a write command, then it checks 71208 whether the disk been

idle from read commands for x milliseconds (x could be 30 for example). (Alternatively, can use other parameters for idleness to determine idleness for the purpose of writing, such as mouse movement, etc.). If idleness for x milliseconds is determined, then a segment of the write queue (e.g. 500KB) is written from the queue to the storage device.