Using Web10G for Service and Network Diversity Exploitation and Failure Recovery

Filipe Carvalho and José Legatheaux Martins

CITI and DI-FCT, Universidade Nova de Lisboa, Quinta da Torre, 2829-516 Portugal
Emai: f.carvalho@campus.fct.unl.pt, jose.legatheaux@fct.unl.pt

***Abstract*—Diversity and redundancy are key properties at the heart of the Internet performance and reliability. However, traditional network and transport interfaces prevent upper layers from directly exploiting them. In order to allow an immediate, ready-to-use, exploitation of diversity and redundancy, we introduced a way of supporting multi-path and multi-server logical connections among a client and a service. The proposal is based on a programming pattern and a run-time support implemented using the Web10G Linux kernel patch, which allows user level access to the kernel TCP variables and performance measures characterizing each TCP connection. The resulting framework allows applications to explore network and service diversity for increased performance and fault recovery experiments.**

I. Introduction

Diversity and redundancy are key properties at the heart of the Internet performance and reliability. Despite this, in general, the interfaces of the different layers (*e.g.*, IP, TCP, HTTP, ...) prevent upper layers from directly exploiting these properties of the different Internet subsystems.

At the network level, choosing alternate paths could be a way to get different qualities of service or using different providers, for example, via client devices with several interfaces and IP addresses, or services connected to different providers and also accessible via different IP addresses. The identity / location separation approaches [1] (*e.g.*, LISP[2]), proposed to enhance the Internet routing and addressing scalability, also call for the use of identity addresses that may be mapped to different locator IP addresses. Some authors, *e.g.*, [3], proposed the generalization of the visibility of addressing alternatives to enhance diversity access.

At the transport level, multiple local interfaces and remote IP addresses are already being considered: Multi-Path TCP [4] is a new version of the traditional TCP protocol that leverages the simultaneous usage of different IP addresses for performance enhancement, and SCTP [5], [6] is a transport protocol that also supports the usage of several interfaces and remote IP addresses to support multi-homing and roaming.

At higher layers, most critical or popular applications (*e.g.*, DNS, content-distribution, mass communication and collaboration, social networks, ...) are implemented using service replication. Many of these systems use automatic redirection of a client to the closest (or preferred) replica. In general, this redirection only takes place at service connection time, using modified DNS servers or HTTP redirections.

Traditional solutions to support diversity and redundancy are based on control planes completely under control of providers, fully opaque to end-systems: routing control planes and content-distribution networks subsystems used to load-balance clients. Nevertheless, there are also many success stories were end-systems take control of a significant part of the control plane. The most successful one is the congestion control system of the Internet, mostly based on the end-systems implementation of TCP which dynamically adapts the emission rate to the status of the network. Other prominent examples are the P2P content distribution systems and more recently the distribution of video content by the so-called Dynamic Adaptive Streaming over HTTP (DASH) [7], [8].

We think that there is some room for more experiments allowing applications to explore network and service diversity for increased performance and fault recovery. For this purpose, and by hypothesis, we consider that network and service alternatives may be accessible using different IP addresses (local and remote). Also, instead of considering new transport protocols, we want to experiment and assess the alternative of only using old ones, namely TCP. Finally, we do not want application code to deal with the complexities of having to directly deal with fault detection and network performance comparisons. Since TCP philosophy and implementation rely on an considerable set of performance statistics, we want to base these experiments in those pre-existing performance indicators.

To this end, we propose a way of exploring multi-path or multi-server TCP-based logical connections among a client and a server, or a service, via an application-level programming framework, to ease network and service diversity exploitation and fault recovery. This functionality is made available through a Java object that supports a logical connection among a client and a server or a service. This object, we dub *NChannelSocket*, can exploit alternative pairs of IP addresses to connect a client to a server or a service. Used in conjunction with an easy to understand programming pattern, it facilitates the development of client / server TCP interactions exploiting path diversity between a client and a server, or among a client and several different but idempotent-equivalent servers.

The goal is to provide a reasonable general-purpose programming pattern and a supporting framework that can be integrated in client / server applications to leverage interface, network and service diversity. In what concerns performance

evaluation and comparison, it relies on parameter estimation that TCP already performs at kernel level. These kernel statistics are ready available in Linux through the Web10G Linux kernel patch [9] and may be easily made available in other operating systems in the future.

In the next two sections we present the proposal and its implementation. A certain number of experiments and the analysis of their results are the object of section IV. Related work is discussed in section V. A discussion of the proposal and future work are both treated in section VI. The paper ends by presenting some preliminary conclusions of this experiment in section VII.

II. Programming Pattern and Run Time Support

The goal of this work is to develop and evaluate a programming pattern [10] and a monitoring runtime support, both geared towards building distributed request / reply applications that explore network and service diversity. As already stated, diversity must be available by means of several (local and remote) IP addresses and we also want to leverage TCP statistics already being collected by the kernel.

The proposal is built around a kind of logical client / server connection, mainly equivalent to a new type of socket, called *NChannelSocket*. At initialization, this object receives several local and remote IP addresses, and establishes a TCP connection among the client and a server using one pair of the available addresses.[1] Each different remote IP address should represent the same server or a server representing the same replicated service. When several servers are used, it is assumed that the service has an interface based on idempotent operations and that the different servers are semantically equivalent. This property is inherent to many services interfaces supported atop of HTTP, a popular solution these days.

During the client / server interaction, if the current TCP connection fails, or a more performant one seems to be available, an alternative TCP connection can be used. Thus, a *NChannelSocket* represents a logical multi-path connection to a server or a logical multi-path connection to a set of semantically equivalent servers representing the same service.

Typical applications of the proposed mechanisms are long client / server interactions, built around a TCP-based request / reply mechanism, to transfer, in several interactions, medium to large volumes of data (*e.g.*, a large file, a stream or a list of related files). The code sketch below is an example of the utilization of the proposed mechanism.

```
NChannelSocket ncs = new NChannelSocket(IP addresses, port, ... );
do {
      try {
         Socket s = ncs.getCurrentSocket();
         // one or more request / reply interactions
            writeRequest ( s, request ) ;
            reply = readReply ( s ) ;
            processReply ( reply ) ;
         if( ! ncs.bestSocketBeingUsed() ) {
             ncs.swithConnection() ;
         }
      }
      catch(Network or Server Exception e) {
             ncs.swithConnection();  }
} while( ! done );
```

The programming style subjacent to the programming pattern corresponds to a series of requests / replies, at the end of each one the client has the choice to follow, or not, a possible connection switching recommendation. See the example in the listing above. To begin with, a *NChannelSocket* is instantiated. Then a loop performs a series of request / reply transactions each using the currently available connection. At the end of each transaction (or several of them) a call to method *bestSocketBeingUsed()* (optionally) asks if a better connection seems to be available and if it is the case, a call to the *switchConnection()* method asks the switch to that better connection. An exception also forces switching the connection (the exception catch and treatment). For the sake of simplicity we do not show the treatment of all other exceptions the object *NChannelSocket* can rise (*e.g.*, no connections available) or other completion conditions.

The *NChannelSocket* supporting runtime includes mechanisms to compute the past performance of the TCP connections made (we called it "history") as well as the performance of the connection currently in use, and also to probe the other available alternatives. Based on these statistics, a recommendation is made available to the application when the *bestSocketBeingUsed()* method is called.[2] The initialization phase as well as the *swithConnection()* method also relies on these statistics to automatically choose and open an alternative connection.

Connection switching is always controlled by the application programmer since it only takes place when it explicitly asks the switch. Therefore, the application programmer has full control over when a potential change of the used server can take place. This may be important for the application and its application logic. Naturally, the application designer also has to define what is a transaction between the client and the service. In most situations where a *NChannelSocket* presents a suitable solution, the transaction is equivalent to a get of a block or a chunk (of a file, or a stream, ...). The size of that block, or set of blocks, must be chosen since it may impact the performance as well as the periodicity with which the method *bestSocketBeingUsed()* is called. In general, between each two calls of the method the connection in use should have the opportunity to attain a steady state (the implementation refrains from recommending a too early switch). Moreover, for short interactions with a server, it may be unreasonable to switch the connection since this introduces at least one extra round-trip time.

In what concerns communications fault recovery, as we will discuss in section IV, the time taken to recover from a broken connection is dependent of the value of the timeout used to limit the time taken to read or write using the current socket. As we rely on TCP, this timeout value can also be controlled

[1] In fact each connection is characterized also by the ports it uses, however we omit that each IP address has a port associated and, for simplicity, consider the TCP connection only characterized by the local and remote IP addresses.

[2] In case of a positive switching recommendation, the method also informs the caller of the nature of the recommendation, see the next section.

and set by the programmer using the TCP sockets provided methods.

III. Implementation

We implemented the *NChannelSocket* abstraction as a Java object encapsulating the runtime framework interface. It relies on a kernel patch (*Web10g* [9], version 3.0, configured on a Linux system with kernel version 3.0.16) that allows access to kernel-level TCP performance statistics, normally hidden from end users, through a kernel *Java Native Interface* (JNI) that we also developed. These statistics include *Round Trip Time* (RTT), *Retransmission Timeout* (RTO), packets received / sent / lost, transfer performance figures, time passed since the connection started and many other attributes.

As previously discussed, we provide support for connection switching from a worst or broken connection to a more viable one. To support this switching we rely on: performance collection, probing and exceptions.

Performance collection is continuously performed using the *Web10g* interface. In every 250 ms the JNI method is called and a new set of measurements is taken out from the kernel. We maintain three working sets of collected data at any given moment. *Last* set, which represents the last measurements taken; *Recent* set, which is the aggregated values of the last ten measurements collected; *Past* set, which represents the updated history of all connections made between any pair of addresses. The later set is saved to a file every 5 seconds (*i.e.*, every 20 samples since 250ms * 20 = 5 seconds). If the values collected are not mean values (*e.g.*, congestion window size) the *Recent* and *Past* sets are always updated to show the most recent mean values. If the values are mean values by themselves (*e.g.*, *SmothedRTT*) the *Recent* set displays the latest values (they are already a mean for this connection), but in the *Past* set the values are saved with a weighted mean (*e.g.*, this connection RTT has less weight than all the past RTTs made with this pair of addresses).

Probing is done periodically by opening, without any data transfer, the alternative connections and measuring the RTT of the probed path / server. If another better connection seems to be available (based of the measured RTT), that will be reported when a call to *bestSocketBeingUsed()* runtime method is made.

Exceptions rise up to the application when a critical error occurs within the connection, either by a timeout or by a link failure. In this case the use of a new pair of addresses is mandatory and a new connection should be started.

Another alternative to switch a connection is when its performance has decayed significantly in correspondence to its past, *e.g.*, an increase in packet loss or a suddenly decrease in throughput.

All the switching logic is intertwined in the method *bestSocketBeingUsed()* that, at each call, checks if the connection is working properly (no loss in performance) and, if not, uses the information from the probing to make a recommendation. Even if the current connection seems to continue to behave properly, inspired from popular P2P systems, we have also implemented a pseudo-random decision method that risks changing connections with low probability if we cannot infer for sure if there is a better path, or with higher probability, if it clearly seems that better alternatives are available. The core of the *bestSocketBeingUsed()* method logic is the following:

```
if (tooRecentlyInvoked() || ! grownUpConnection(currentConn)
            || ! availableAlternatives(currentConn)) {
   return 0;  // do not switch recommendation
}

if (senderDependent(currentConn)
         && isWindowInRecoveryState(currentConn)) {
   if (highSenderPacketLoss(currentConn,pastCon)
         || receiverThroughputDecayed(currentConn,pastConn)) {
      return 1; // recommend switch type 1
      }
}

if ( receiverDependent(currentConn)
         && senderThroughputDecayed(currentConn,pastConn)) {
   return 2; // recommend switch type 2
}

if (isThereABetterPath(currentConn))
   switchProb = HIGHPROB;
   else switchProb = SMALLPROB;
double random = Math.random();
if(random < switchProb){
   return 3; // recommend switch type 3
}
// otherwise, do not switch recommendation
return 0;
```

The *currentConn* and *pastConn* parameters represent the current connection in use and it's past in the *Past* set, respectively. The method returns a "no change" suggestion if it has been called recently, the current connection is not grown up (*i.e.*, has, in the current implementation, less then 100 RTTs or less then 5 seconds) or there are no available alternatives.

If the client is sending more data than receiving (thus the connection is sender dependent), we check if the congestion window is in the recovery state, i.e. if the size is equal or less then $2 * MSS$ (Maximum Segment Size), and if the sender packet loss (*i.e.*, the client packet loss) or the receiver throughput (*i.e.*, the throughput from the other side of the connection) has recently decayed. Otherwise, if the client is receiving more data than sending (thus the connection is receiver dependent), we check if the sender throughput (*i.e.*, the throughput from the other side of the connection) has recently decayed. Both return types 1 and 2 signals the application that the performance has decreased. Return type 3 is used to signal the application to take a risk, with higher or lower probability, as explained before.

When the method *switchConnection()* is called, the best alternative is selected using *Smoothed RTT* as the main criterion if past performance is unknown. After the creation of a *NChannelSocket*, the first connection is chosen randomly or, if history data is available, based on the past performance.

In summary, *NChannelSockets* provide: connection initialization; connection history collection where the past performance of all TCP connections made are saved in a file; probing of connections; choice of initial pair of addresses, through connection history if available, or randomly choosing one pair of the available addresses to start with; and choice of posterior pairs of addresses when the current connection compares unfavorably, or if it broke. Together, these functionalities are used

TABLE I
CHARACTERISTICS OF THE LINKS USED IN THE FIRST TEST SCENARIO.

Link	Capacity	RTT	Packet loss
Worst-path	700 Kbps	15 ms	1%
Middle-path	900 Kbps	15 ms	1%
Best-path	1000 Kbps	5 ms	1%

to test and recommend the switching between connections to maintain the transfer active and as performant as possible.

IV. EVALUATION

To evaluate the proposal we developed an application that downloads a file, via a sequence of block transfers using HTTP Partial Requests and uses the programming pattern and framework presented above.

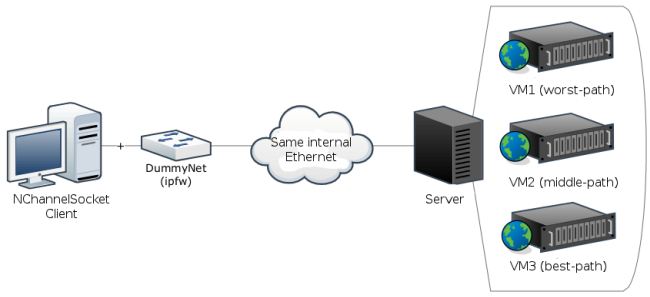

Fig. 1. First test scenario

The first test scenario was configured as follows: a client with one local address and three remote servers with three different addresses (three virtual machines in one server rack) but replicating the same 20 M Byte object. Client and servers are connected to the same switch as shown in Figure 1.

With the help of *Dummynet* [11] we simulated three different paths to the servers (see Table I). The three paths had no significant performance differences since we intended to test the ability of the framework to compare their characteristics and elect the best one.

Due to the emulated link capacities chosen, the test scenario is not fully representative of the variations in path performance in the real Internet. This was done to make the test scenario simple. The links that we emulated have characteristics that are no longer common place in the Internet given that, for example, higher throughputs are becoming a more common scenario in home broadband access settings. Our choice rests on the fact that we wanted to emulate a long file transfer, and instead of using files with large sizes, again because of the time resources available, we opted for a smaller file, with slower transfer rates.

Control tests were used to devise how the different parameters influenced the switching recommendation mechanism and the total transfer time. For example, we adopted a block size of 512 K Bytes, since increasing it does not decreased the total transfer time, while when decreasing it, performance suffered. With the best-path, this block is transferred in 4 seconds, what

TABLE II
JAVA SOCKET, *NChannelSocket* AND *WGet* TRANSFERS COMPARISON IN SECONDS.

	Java Socket	*NChannelSocket*	WGet
Mean	165.59 ± 0.42	165.83 ± 0.62	165.30 ± 0.82

TABLE III
ONE PATH *NChannelSocket* TRANSFERS COMPARISON IN SECONDS.

	Worst-path	Middle-path	Best-path
Mean	245.41 ± 2.40	195.47 ± 3.36	165.84 ± 0.63

corresponds to a not very tight control of the performance but allows better TCP performance estimation. We will return to this point below. We also concluded that using the pattern with one only available path corresponds to almost the same performance we could get with a traditional Java Socket or even while using an optimized Linux utility like *wget* (allways using the same path), as can be seen in Table II. Thus, with this block size, the pattern and the framework do not penalize a file transfer in the case where no alternatives are available.

We also computed the time needed to transfer the file over each of the available paths, *i.e.*, when one only path was available to the *NChannelSocket*, see Table III.

Control tests, as well as all tests presented below were repeated 20 times, giving us a good mean and standard deviation values to support our conclusions. All the tests presented from now on were done using simultaneously all the three paths.

In test 1 no previous history was available, while test 2 was made taking in consideration the measured past performance of the paths. These tests were useful to see if our framework starting from a random path eventually changes to the fastest available one (in test 1), or if starting from the better one (test 2) it stayed with that connection, thus testing positively the advantage of using information based on the past performance of the paths. Results for tests 1 and 2 can be seen in Table IV. In test 1 the client always switched to the best path while during test 2 the client always stayed with the best path.

Test 3 was made to check if when the best connection available (the one being used) fails persistently, the framework can chose another one, and how long it takes to switch to an alternate path. Test 4 checks if the framework can detect a decrease in the performance of the connection being used, how long it takes to detect it, and how long it takes to switch to another, better, connection. Results for tests 3 and 4 can be seen in Tables V and VI. The "switch at" column represents time after the best-path failure or performance update.

These two tests took noticeably more time to transfer the file, because faults and performance problems were introduced. After 20 seconds of starting the transfer, we completely blocked the path in use (Test 3) or decreased it's performance significantly to 100 Kbps (Test 4). With test 3, when a failure is detected, after a TCP enforced timeout of 1000 milli seconds, the framework immediately provides an alternate connection. In this case, the available paths were worst than the one

TABLE IV
TESTS 1 (NO HISTORY) AND 2 (WITH HISTORY) - TOTAL TRANSFER TIME IN SECONDS.

	No History	With History
Mean	171.04 ± 1.36	167.68 ± 0.86

TABLE V
TEST 3 (PERSISTENT FAILURE) - TOTAL TRANSFER TIME IN SECONDS.

	Persistent Failure	
	Total time	Switch at
Mean	202.52 ± 14.55	1.27 ± 0.34

TABLE VI
TEST 4 (DECREASE IN PERFORMANCE) - TOTAL TRANSFER TIME IN SECONDS.

	Decrease in performance	
	Total time	Switch at
Mean	355.65 ± 51.34	178.38 ± 62.36

that failed, hence the increasing in transfer time. Figure 2 represents the throughput in our application for the channel being used at a given moment. This chart shows clearly that, after 20 seconds, the throughput for the *best-path* dropped to zero, i.e. a link failure occurred. After that failure another path was chosen (*middle-path*). After a while even another path was chosen (this time the *worst-path*). This path was chosen because our system relies heavily on *Smoothed RTT* when no previous transfer performance is available, and the *worst-path*, despite having worst throughput, was with better measurements of *Smoothed RTT* at the time.

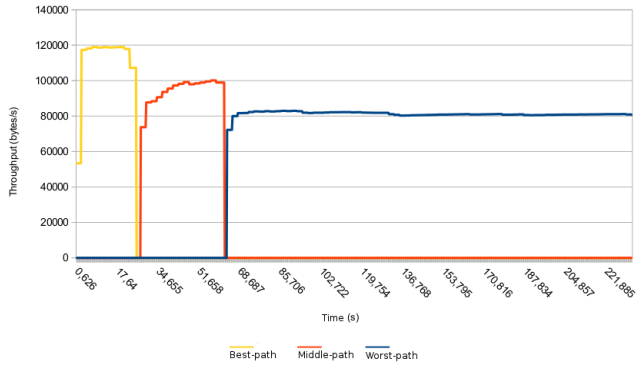

Fig. 2. Test 3 - Transfer Example

Test 4, on the other hand, takes longer, since the parameters used to detect when a path decreases its performance revealed hard to fine tune. In fact, in some of the 20 performed tests, the framework took longer to react.

Now we turn to a different scenario intended to test a home client with two interfaces (wired and wireless) accessing to a remote server, over the public Internet, to download a file of 150 MBytes. The client started by using the path over the wired link. After 15 seconds this interface was shutdown, an exception arose after the timeout and a new connection was opened over the wireless link. Results can be seen in Table VII. The first column shows the results when only the wireless link was used, the second when only the wired link was used and the third shows a scenario where the transfer started over the wired interface and after the fault of that interface switched to the wireless one. Since the switch to the alternative interface is quick, only the block being transferred while the fault arose was lost, what corresponds in average to a very short period. Recall that the block size is 512 K Bytes and the file had 150 M Bytes, thus each block was transferred in less than 0.5 seconds. The end user would notice almost no difference.

TABLE VII
TEST 5 (WIRED LINK FAILURE) - TOTAL TRANSFER TIME IN SECONDS.

	Wireless (s)	Wired (s)	Link failure (s)
Mean	119.47 ± 2.24	115.93 ± 0.57	117.05 ± 0.34

Although more varied tests with different link characteristics could be performed, the presented ones are significant to highlight the functionality and behavior of our proposed framework.

V. RELATED WORK

Our proposal adheres to a simple evolutionary approach by using several TCP connections and alternatively switching among them. Other evolutionary approaches are possible. Multi-Path TCP [4] leverages path diversity using all paths simultaneously, however it cannot support server diversity and requires the adoption of a new version of TCP. SCTP also supports the usage of several local and remote IP addresses and there is a very active research on leveraging this feature to support transparent handover and multihoming [6], since the original standard only used it to support backup paths. Both transport protocols do not support the notion of a connection between a client and a multi-server service, maintain diversity hidden from application programmers and represent a significant departure from current stacks, something we would like to avoid.

Leveraging diversity has been a recent trend that most P2P systems are engaged in. Many techniques have been developed to allow peers to choose, among the several available alternatives, the ones that are most effective. This optimization can be solved by the peers alone or with the help of some extra information, made available directly or indirectly by (network or content) providers. Paper [12] presents a survey of the issues and the solutions. We introduced a general purpose framework to make available at application level the information TCP already collects at the host kernel level. Our search for more generality and application neutrality, as well as the leveraging of the kernel TCP implementation are different, but can be extended if providers provided hints on the quality of their paths.

Most critical Internet services (*e.g.*, DNS, CDNs), applications and subsystems use replication techniques to replicate data and to redirect clients to the best available replica.

We propose an approach that empowers clients to choose themselves the most suitable server.

Using local different addresses closely related to different interfaces is now common place. Visibility of different server IP addresses is not yet widespread. However, the so called "Identity / Location Separation" approaches are based on the usage of identity addresses, mapped to different location addresses [3], [2]. Location addresses are closely related to network properties and also denote paths, an idea that the work described in [13] took to its limits. An upper level service (*e.g.*, DNS or a LISP mapping server) can provide the extra indirection between a server or a service and the several addresses that may be used to access them. *NChannelSockets* builds on this eventual address diversity to explore network diversity.

VI. Discussion and future work

Our approach provides the same abstractions and mechanisms to be used in two different scenarios: edge exploitation of network diversity and application exploitation of service diversity. Concrete experiments made to explore both scenarios should be performed. Perhaps, these experiments will recommend the separation of the programming patterns (and the support mechanisms) to be used in these two different contexts.

Service diversity exploitation requires more application engagement. For example, it would be challenging to use the proposed approach to implement a video streaming client capable of automatically adapt the video resolution to the available network performance, as current dynamic adaptable streaming over HTTP clients do, but also simultaneously exploiting alternative servers delivering the same video, to find the best one. This kind of application requires a close control of the switching moments and of the transactions between clients and servers as our proposal allows. Nowadays, adaptable streaming is being scaled to millions of costumers, by simultaneously leveraging several content distribution networks. A provider side control plane is already used to direct clients to a specific data center. This control plane could be improved with client side options.

Multi-Path TCP and SCTP are both well suited to leverage transparent network diversity exploitation in a stable client server relationship. In this scenario, it seems that both protocols are better alternatives than our proposal. However, they prevent any control of the application over the criteria used to choose paths. Our framework can be easily extended to support other criteria to rank connections. For example, connectivity price can be an extra parameter to be considered when deciding to switch connections. Our proposal should also be extended in this direction. Multi-Path TCP or SCTP seem less suited to support that kind of experiments.

Finally, TCP interprets missed acknowledges as originated by congestion episodes, not by path failures, and waits a long period before declaring a TCP connection as broken. Therefore, we use user level timeouts to detect path failures. If we used very short timeouts (*e.g.*, 100 milliseconds), we may wrongly interpret these exceptions as path faults, and introduce path switching instability. Thus, better ways to distinguish path failures from path congestion are needed and should be researched. However, this problem probably stresses the limits of a fully client centric fault detection method. In fact, if the RTT of a connection is significant, the detection by the client that the path broke will take several rounds, and will be impossible in the sub second range.

VII. Conclusions

In his paper we have presented a programming pattern and a support framework whose joint usage empowers a client of a service to explore network and service diversity and mask faults. The programming pattern is very simple and easy to use and it successfully allows the programming of a client of a replicated service, or of a server reachable by different paths, to transparently explore several servers and network paths for performance enhancing and fault masking. The proposal is reasonably application neutral, is based on the Web10G Linux kernel-level patch and interface and provides user processes with means to explore and choose among several alternative TCP connections between the client and a potentially replicated service. No changes of the traditional TCP/IP stack or of upper level protocols were required.

References

[1] D. Jen, M. Meisel, H. Yan, D. Massey, L. Wang, B. Zhang, and L. Zhang, "Towards A New Internet Routing Architecture: Arguments for Separating Edges from Transit Core," in *Seventh ACM Workshop on Hot Topics in Networks (HotNets-VII)*, 2008.

[2] D. Meyer, "The locator identifier separation protocol (lisp)," *The Internet Protocol Journal*, vol. 11, no. 1, March 2008.

[3] V. Kafle and M. Inoue, "Introducing multi-id and multi-locator into network architecture," *Communications Magazine, IEEE*, vol. 50, no. 3, pp. 104 –110, march 2012.

[4] A. Ford et al., "Architectural guidelines for multipath TCP development," Internet Engineering Task Force (IETF) - RFC 6182, March 2011.

[5] E. Stewart, R., "Stream control transmission protocol - proposed standard," Internet Engineering Task Force (IETF) - RFC 4960, September 2007.

[6] T. Wallace and A. Shami, "A review of multihoming issues using the stream control transmission protocol," *Communications Surveys Tutorials, IEEE*, vol. 14, no. 2, pp. 565 –578, quarter 2012.

[7] T. Begen, A. Akgul and M. Baugher, "Watching video over the web: Part 1: Streaming protocols," *Internet Computing, IEEE*, vol. 15, no. 2, pp. 54–63, march-april 2011.

[8] S. Akhshabi, A. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," *ACM MMSys*, vol. 11, pp. 157–168, 2011.

[9] Web10G Project. http://web10g.org, 2011.

[10] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[11] M. Carbone and L. Rizzo, "Dummynet revisited," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, pp. 12–20, Apr. 2010.

[12] J. Dai, F. Liu, and B. Li, "The disparity between p2p overlays and isp underlays: issues, existing solutions, and challenges," *Network, IEEE*, vol. 24, no. 6, pp. 36 –41, nov-dec 2010.

[13] X. Yang, D. Clark, and A. Berger, "Nira: A new inter-domain routing architecture," *IEEE/ACM Transactions on networking*, vol. 15, no. 4, pp. 775 –788, aug. 2007.