DL^3: Adaptive Load Balancing for Latency-critical Edge Cloud Applications

Prashanth P S — *IIT Hyderabad*
Ranjitha K — *IIT Hyderabad*
Ankit Sharma — *IIT Hyderabad*
Arjun Temura — *IIIT Delhi*
Rinku Shah — *IIIT Delhi*
Praveen Tammana — *IIT Hyderabad*

***Abstract*—On-premise edge cloud provides opportunities to enable ML-based latency-critical services to resource-constrained end devices. The edge services are deployed as loosely coupled microservices using cloud orchestrators like Kubernetes, and a load balancer distributes requests from an upstream microservice instance (client) across many downstream microservice instances (servers). However, in a shared environment, transient and sporadic delay events are common due to contention for host and network resources (*e.g.*, high load on servers, high network queuing delays). To meet low latency requirements of edge services, the load balancer should *quickly adapt* to such delay events and *adjust* routing decisions (*e.g.*, pick the best downstream instance among all). In this paper, we propose DL^3, a distributed load balancer (LB) that quickly adapts to server load and network queuing delays by adjusting routing decisions so that the requests are forwarded to the best possible servers. The key idea is to enable LB with visibility into both servers' load and transient delays on network paths toward the servers. We prototype DL^3 on a Kubernetes-managed edge cloud cluster and evaluated its performance for a latency-sensitive ML-based object detection service. Our preliminary results show that DL^3 improves tail response time by 33% compared to the state-of-the-art load balance mechanism.**

I. Introduction

For compute and memory-constrained IoT devices, edge cloud enables access to the latest and most powerful compute, storage, and services deployed in the cloud [1], [2]. The benefits include (1) high-speed responses for latency-sensitive applications, (2) security and data privacy by processing sensitive data locally, and (3) optimization of bandwidth between edge and central cloud. This paper focuses on load balancing problems at the edge cloud that degrade response time for latency-critical applications. More specifically, we consider latency-sensitive ML-based object detection service deployed at the edge cloud, orchestrated by Kubernetes-Istio [3], [4], for various IoT [1], [5], [6] and autonomous navigation [7] applications; the end devices send video stream to the edge cloud, and the object detection service responds with the objects detected.

Service meshes (*e.g.*, Istio [4]) provide a load balancer that sits next to each microservice instance (client-side), intercepts outgoing requests, and load balances the requests among instances of the downstream service (server-side). A load balancer (LB) at an upstream microservice instance (client) aims to minimize the response time by quickly adapting to dynamic events (*e.g.*, high load on servers, high network delays) and pick the best possible downstream instance (server) among all the instances. To meet this objective, the client-side LB should be aware of two types of information. First, it should know the current aggregated load on all downstream instances [8]–[10], that is, the current in-flight request count at each downstream instance (sum of the number of active requests from all upstream LBs). The server load information helps LB to come up with better routing decisions. Second, the LB should be aware of sporadic and transient delays on network paths toward the downstream servers [11]–[14] so that the LB routes requests to servers with path delay under an acceptable limit and meet service-level agreement (SLA).

In this paper, we design DL^3, a (D)istributed (L)oad balancer with awareness on Network (L)atency and Server (L)oad. Our key idea is to enable client LB with visibility into two dynamic events: (1) server load at the application layer and (2) transient delays on network paths toward the servers. By doing so, LB has a view of the best of both worlds and quickly adapts to dynamic events by routing requests to the best possible servers while ensuring that the compute and network resources are not underutilized. DL^3's design is based on the principle that the best source of data on a server load is the server itself, and the best source of data for network delays is the client. DL^3 provides this data to client-side LB; it enhances server report-based layer-7 load-sharing mechanisms with the status of network path delays observed by a client instance at layer-4. DL^3 tracks this data and provides a list of healthy and unhealthy servers so that the client-side LB routes requests to healthy servers and reduces tail response latency.

II. Background and motivation

ML-based object detection. In this paper, we focus on ML-based object detection (yolo [15]) on video streams from IoT devices (*e.g.*, robots) with low-latency requirements (see Fig. 1). In our experiment setup, one application container represents one IoT device, and each application container streams video frames as HTTP requests. These requests are intercepted by the sidecar (proxy) container's LB which distributes the requests among all downstream object detection servers (containers); these servers detect objects using GPU compute and respond with object coordinates (bounding boxes).

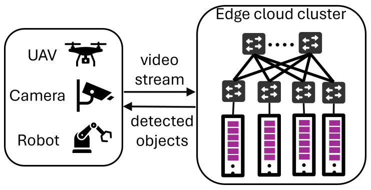


Fig. 1: **ML-based object detection using compute at the edge cloud**

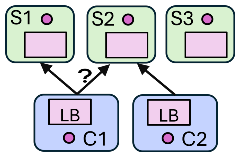


Fig. 2: **Local in-flight request count is misleading**

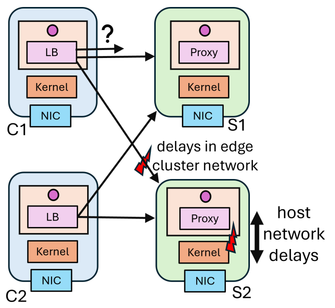


Fig. 3: **Local LB lacks view on network delays**

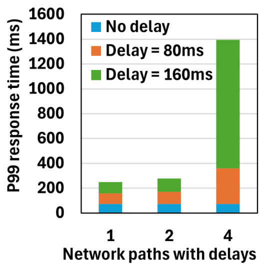


Fig. 4: **P99 response time increased significantly (1.2x-14x) due to transient delays in the host network stack**

Loadbalancing using P2C and least connections. Service meshes like Istio [4] use envoy proxy as a sidecar. By default, Istio uses the most popular least requests load balancing policy [10], which randomly selects two healthy downstream nodes using the power two choices (P2C) [16] and greedily selects the node that has the least number of active requests (Join the shortest queue) based on its local view. Join the shortest queue (JSQ) in combination with P2C addresses the herding problem of the least connection first approach [8].

Local load balancers' view on in-flight request count can be misleading. In a cluster of LBs, a single load balancer's local view (client-side) on a server's in-flight request count could differ from the actual request count (see Fig. 2) at the server because the server gets requests from all clients' load balancers. To avoid this problem, an upstream node (client) should know the aggregated load on each downstream node (server) so that the upstream nodes' LB routes requests to a server with the least load. Otherwise, the load balancer could overload the server. For instance, Fig. 2 illustrates a case where server S2 is overloaded when client C1 tries to distribute the load based on its local view, leaving S3 underutilized.

Network delays affect response time. A request from a client pod to a server pod traverses multiple entities and network components at the host level and data center network. The entities include the request's source node, the destination node, and the underlying network (switches and routers). Within each node, a request/response traverses multiple components in the host network stack: NIC, root namespace, and container namespace. Transient and sporadic delay events (*e.g.* queuing delays) at any one or more of these components inflate request completion time (RCT). Recent studies observe that application-level RCTs do not meet the SLAs due to packet-processing delays (10s-100s of millisecond) at these components [11]–[14]. Since node resources are shared across multiple tenants (containers), variable queuing delays are observed due to unexpected CPU scheduling [12]–[14], head-of-line (HOL) blocking [11], [13], slow read from the NIC Rx buffer at the host level [12], policy enforcement as load increases [17], [18], and server livelock [13], [19].

Local LBs lack a view on network delay events. The current least connection approach at the client LB does not account for such delay events on network paths toward servers before routing requests. For instance, as shown in Fig. 3, consider two downstream servers (S1 and S2), each with the same local in-flight request count at the client LB (C1), and server S2's requests are experiencing network delays. When a new request arrives at C1's LB, routing the request to server S1 may process the request faster than S2 because S1 receives request data without experiencing network delays. To avoid this problem, the client LB should be aware of network delays on paths toward servers and route requests to servers based on their path delay status.

To understand the impact of network delays on response time, we conduct an experiment on a Kubernetes cluster of pods with Istio's envoy proxy. The setup has 8 client pods and 8 server pods. We introduce synthetic network delays (using Netem [20] tool) on up to four paths, one each towards four servers. Fig. 4 shows 99 percentile response time without and with delays. With a network delay of 80ms that lasts for 100ms on one path, object detection response time is increased to 1.2x. It is 4x with delays on four paths toward four servers. With 160ms queuing delay, P99 response time is increased by 1.25x and 14x, respectively. This motivates the need for a network delay-aware load balancer that quickly adapts to dynamic delay events and picks the best server.

III. Design

Our key idea is to enable client LB with visibility into two dynamic events: (1) server load at the application layer and (2) transient delays on network paths toward the servers. By doing so, LB has a view of the best of both worlds and routes requests to the best possible servers while ensuring that the compute and network resources are not underutilized.

We consider the following design principle. The best source of data on a server load is the server itself, and the best source of data for network delays is the client.

A. Server shares the current load with clients

Monitoring. There are two ways to know servers' load. First, all downstream servers should be actively polled for their current in-flight count. This is expensive for a large cluster because it requires N LBs to poll M servers every few milliseconds. Second, every server piggybacks the current load to the response, and the client LB tracks the responses and finds the servers' load. In the second approach, client LBs are more up-to-date when the request rate is high. We choose the second approach because our object detection use case will

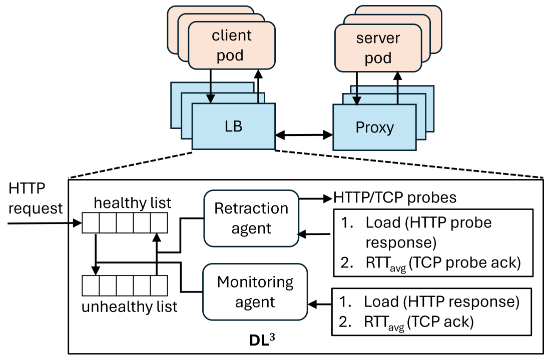


Fig. 5: **DL^3 system overview**

have a continuous video stream (frames per second), so the request rate is high.

As shown in Fig. 5, the server's proxy intercepts HTTP response and tags the current in-flight request count (load). We configure the client proxy with the maximum number of active requests (T_{load})[1] a server can handle to meet performance SLOs. The proxy computes load based on the current in-flight request count. On response arrival, a monitoring agent at client LB parses the HTTP response, if the load $< T_{load}$, the agent puts the server in the healthy list; otherwise, it puts the server in the unhealthy list.

Retraction. To avoid underutilization of server resources, a server in the unhealthy list should be moved to the healthy list at the right time, that is, right after the load on the server is below the threshold. we choose a proactive approach where the client LB agent sends HTTP probe requests periodically to the unhealthy servers and updates their status. This enables the client LBs to be up-to-date on the load on unhealthy servers. The probe rate is proportional to the number of unhealthy servers at a single LB. Probes are sent to an unhealthy server as long as it is overloaded. The probe overhead is generally low when the requests are distributed well.

To keep probe overhead low, before sending the next probe, the client LB agent waits for at least the average RCT times the number of active requests (server load observed in the previous probe) above the T_{load}. If the current load on an unhealthy server is less than the T_{load}, it is moved to the healthy list so that the client LB routes requests to the server.

B. Client LB monitors TCP RTT to understand network delays

RTT monitoring. To find network path delays, DL^3 monitors the weighted average RTT (RTT_{avg}) of TCP packets of in-flight requests. If the RTT_{avg} of in-flight requests exceeds T_{RTT}[2], the associated server is moved to the unhealthy list; this step avoids routing requests to servers with high network delays, especially when the in-flight requests are in progress. Despite the delays, the in-flight requests are not rerouted to another server due to the costs involved in breaking the existing connection at the current server and transferring the already sent data to another server. However, the requests that arrive at client LB during these network delay events are routed to healthy servers, thus improving the response time. RTT_{avg} for every new RTT of a TCP packet is calculated as RTT_{avg} = (1-α) * (RTT_{avg}) + α * RTT_{new}, where, α = 0.125.

Retraction. To avoid underutilizing available network bandwidth and overloading healthy servers, servers in the unhealthy list should be moved to the healthy list right after the delay events fade away. We choose a combination of reactive and proactive approaches to know the path's status to an unhealthy server. More specifically, if the in-flight request to an unhealthy server is active, then RTT_{avg} of TCP packets of this request is used to understand the path status (reactive); this avoids sending additional probe traffic to the unhealthy server, especially if delay events fade away before completion of the in-flight request. If the in-flight request finishes and the delay events persist until the end, then we start probing the path using TCP probes and monitor the RTT_{avg} of these probes to know the path status.

If $RTT_{avg} > T_{RTT}$, then it checks for the in-flight request status, and if present, then probes are not initiated. Otherwise, probes are sent periodically and whenever RTT_{avg} is below the threshold, the associated server is moved to the healthy list.

C. Load balance requests

Upon receiving a request from an application container pod (client), the LB in the sidecar proxy follows the P2C technique: it randomly picks 2 servers from the healthy servers' list and routes the request to the one with the least in-flight request count from its view.

IV. Implementation

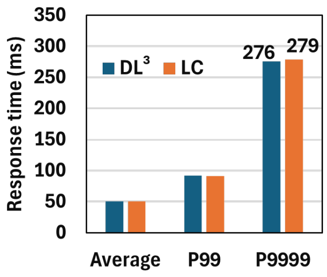
We prototype DL^3 using BLOC servicemesh [21] and instrumented BLOC proxy with DL^3. DL^3 implementation comprises two modules: 1) monitoring agent, and 2) retraction agent. The monitoring agent tracks the server load carried in an HTTP response and average RTT for in-flight requests.

Monitoring servers' load: The proxy is instrumented with a counter so that the number of active requests at the application layer is tracked, and the counter value is tagged to the HTTP response header. On receiving an HTTP response, the client-side proxy parses the response header, removes the counter, and forwards the HTTP response to the client pod.

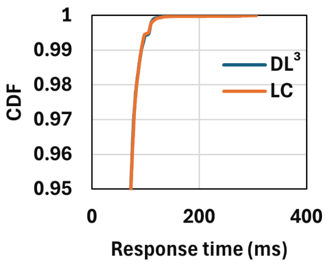
Monitoring network delays: The proxy is instrumented with an always-on RTT monitoring daemon process which uses socket statistics(ss) utility that reads the average RTT of a TCP connection (to a server) from the kernel. We have instrumented the daemon process to read the average RTT value periodically for every 2ms so that transient packet processing delay events that last for 10s of milliseconds are captured. If there are no ongoing requests to unhealthy servers, the client-side proxy uses the hping utility to send periodic probe signals to the server till the average RTT $< T_{RTT}$.

[1]To find T_{load} for a service either profile service latencies using test traffic or collect measurements for real traffic.

[2]To find T_{RTT} for a service either profile delays using test probes or collect RTT samples for real traffic.

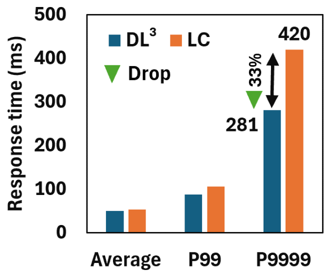


(a)

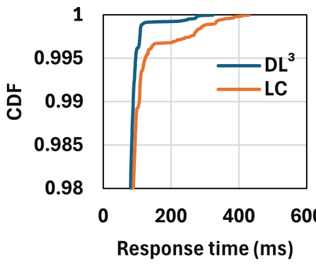


(b)

Fig. 6: **DL^3 response time is on par with LC when there are no network delays.**



(a) **80ms delay lasting 200ms**



(b) **80ms delay lasting 200ms**

Fig. 7: **33% improvement in P9999 response time of DL^3 compared to LC when there are network delays**

V. Preliminary Evaluation

Our evaluation primarily aims to study the improvement in response time by DL^3 compared to LC in the presence of network delays.

A. Experiment setup

We evaluate DL^3 in a Kubernetes cluster with three bare metal servers (one master node and two worker nodes) [22]. We deploy a Yolov5-based object detection service for autonomous navigation applications as a two-layer service chain with an 8:8 client-to-server pod ratio. We deploy Ubuntu pods (client pods) to stream an offline video of vehicular traffic. Server pods running the Yolov5 model perform the object detection and return the detected object with its confidence level. All pods in the cluster are deployed with BLOC [21] proxy as a sidecar running the load balancing logic. We profile the service and empirically fix the threshold value for request count at 10 and RTT as 10ms.

B. Request response latency with DL^3

We study the performance of DL^3 by evaluating against the Least Connections (LC) under two scenarios: (1) without any network path delay (baseline case), and (2) with network path delays. We send video frames from each client pod as HTTP requests at 20FPS (frames per second) for 120 seconds asynchronously. To emulate a network delay event, we synthetically inject queuing delays 80ms at one server pod's veth lasting for 200ms at a frequency of 12 faults per minute using the Netem tool. We monitor the response time at client pods and use the average, P99, and P9999 values of response time as the evaluation metric.

Baseline results. Fig. 6a shows the average, P99, and P9999 response time and Fig. 6b shows CDF for LC and DL^3 without network delays (baseline case). From the results, we observe that DL^3's load-balancing strategy works well and at par with LC when there are no network delays on paths toward servers.

Comparison with the baseline. Fig. 7 shows the average, P99, and P9999 response time for LC and DL^3 with 80ms network path delay lasting for 200ms. From the results, it is worth noting that the average and P99 of LC and DL^3 remain almost similar to the values observed for the baseline case (Fig. 6a). However, compared to the baseline case, the P9999 value of LC increased significantly by up to 50%, whereas the DL^3 P9999 response time observed a minimal increase of 2%. This is because the LC LB technique at the client is unaware of the transient (too short to react) network queuing delays on the path towards a server and forwards subsequent requests to the same server, so the requests end up waiting in the network queue. DL^3 handles this problem by leveraging awareness on path delays, so it picks another server. Fig. 7b shows the CDF for DL^3 and LC with network delays. This highlights the effectiveness of DL^3's load-balancing strategy.

VI. Related work

Load balancing with limited visibility. Load-balancing strategies such as round robin [23], random [24], and P2C [16] are widely used because of their simplicity, but they lack a view on load and network delays, thus suffer from high tail response latency. Prior works [8], [9], [25], [26] provide LB with visibility over the server's load. In [8], the in-flight requests count is piggybacked from the server to the client. In [9], the servers probabilistically piggyback confidence chips, which indicates to the client that the server can handle more requests. However, in the presence of transient and sporadic delay events on network paths, they both suffer from high tail response latency. Congestion control mechanisms [27], [28] take inspiration from [29]–[31] and provide visibility of network path status. These mechanisms enable clients to quickly adapt by adjusting sending rates. But without visibility on the server's load, an LB can pick an overloaded server with good network status, thus inflating response time.

Load balancing with visibility. [14] load balances requests using RCT with Direct Server Return (DSR) in place. The load on a server is approximated using the next triggered request after the client receives the previous request's response. This approach works well for sequential requests, but with asynchronous and parallel requests, the LB cannot assess the server load until the next request arrives. Therefore, it is too slow to react and ineffective as the load dynamics and path status can change quickly.

VII. Conclusion and future work

We propose DL^3, a distributed load balancer for latency-critical services deployed in the edge cloud. DL^3 *quickly adapts* to transient and sporadic delay events in the edge cloud by *adjusting* layer-7 routing decisions. DL^3 LB's awareness of servers' load and network delays on paths toward servers enables routing requests to the best possible servers and improves response time. In our future work, we would like to study the overheads introduced by DL^3. We also plan to compare DL^3 with other approaches in the literature.

Acknowledgement

We thank Yuvraj Makkena, K Shiv Kumar and Harish S A for the helpful discussions and their valuable feedback on this work. This work is supported by the TiHAN IIT Hyderabad, ZF Technologies, National Security Council Secretariat, India.

References

[1] Ajay Kumar Tanwani, Raghav Anand, Joseph E. Gonzalez, and Ken Goldberg. Rilaas: Robot inference and learning as a service. *IEEE Robotics and Automation Letters*, 2020.

[2] What is edge-cloud? https://www.vmware.com/topics/glossary/content/edge-cloud.html.

[3] Kubernetes. https://kubernetes.io/.

[4] Istio. https://istio.io/.

[5] K. Hagiwara, Y. Li, and M. Sugaya. Weighted load balancing method for heterogeneous clusters on hybrid clouds. In *IEEE EDGE*, 2023.

[6] HyunJong Lee, Shadi Noghabi, Brian Noble, Matthew Furlong, and Landon P. Cox. Bumblebee: Application-aware adaptation for edge-cloud orchestration. In *IEEE/ACM SEC*, 2022.

[7] Peter Schafhalter, Sukrit Kalra, Le Xu, Joseph E. Gonzalez, and Ion Stoica. Leveraging cloud computing to make autonomous vehicles safer. In *IEEE/RSJ IROS*, 2023.

[8] Mike Smith. Rethinking netflix's edge load balancing, September 2018.

[9] Ratnadeep Bhattacharya and Timothy Wood. Bloc: Balancing load with overload control in the microservices architecture. In *IEEE ACSOS*, 2022.

[10] Least requests. https://www.envoyproxy.io/docs/envoy /v1.5.0/intro/arch_overview/load_balancing.

[11] Marek Majkowski. The story of one latency spike, November 2015.

[12] Theo Julienne. Debugging network stalls on kubernetes, November 2019.

[13] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, and Timothy Merrifield et. al. How to diagnose nanosecond network latencies in rich end-host stacks. In *USENIX NSDI*, 2022.

[14] Bhavana Vannarth Shobhana, Srinivas Narayana, and Badri Nath. Load balancers need in-band feedback control. In *ACM HotNets*, 2022.

[15] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *IEEE CVPR*, 2016.

[16] Michael Mitzenmacher. Handbook of algorithms and data structures. https://www.eecs.harvard.edu/ michaelm/postscripts/handbook2001.pdf, 2001. Accessed: 2024-07-01.

[17] Mrittika Ganguli, Sunku Ranganath, Subhiksha Ravisundar, Abhirupa Layek, Dakshina Ilangovan, and Edwin Verplanke. Challenges and opportunities in performance benchmarking of service mesh for the edge. In *IEEE EDGE*, 2021.

[18] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, and Xuan Kelvin et. al. Zou. Dissecting overheads of service mesh sidecars. In *ACM SoCC*, 2023.

[19] Jeffrey C Mogul and Kadangode K Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 1997.

[20] Netem. https://srtlab.github.io/srt-cookbook/how-to-articles/using-netem-to-emulate-networks.html.

[21] Bloc github. https://github.com/MSrvComm.

[22] Yuvraj Chowdary Makkena, PS Prashanth, Praveen Tammana, Praveen Chandrahas, and Rajalakshmi Pachamuthu. Real-time object detection as a service for ugvs using edge cloud. In *IEEE COMSNETS*.

[23] Round-robin. https://www.envoyproxy.io/docs/envoy/v1.5.0 /intro/arch_overview/load_balancing.

[24] Random. https://www.envoyproxy.io/docs/envoy/v1.5.0 /intro/arch_overview/load_balancing.

[25] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale rpcs with breakwater. In *USENIX OSDI*, 2020.

[26] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, and Sameer G. et. al. Kulkarni. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *ACM SoCC*, 2021.

[27] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, and Monia et. al. Ghobadi. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[28] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, and Lingbo et. al. Tang. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[29] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: new techniques for congestion detection and avoidance. In *Conference on Communications Architectures, Protocols and Applications*, 1994.

[30] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. Fast tcp: Motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 2006.

[31] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *IEEE INFOCOM*, 2006.