DCSL Project Report **Konstantinos Tsitsimpikos**
Prof. Edward Bugnion Lab Date: 04/10/18
Supv: Marios Kogias Due Date: 11/01/19

Problem Statement

Tail Latency Analysis on Fanout DataCenter Systems.

Modern cloud services require increasingly stricter service level objectives for performance. These objectives are often defined in terms of a percentile of the latency distribution(such as the 99.9th-tile). Moreover, for interactive Web services, requests with a **high fan-out** that parallelize read operations across many different machines are becoming a common pattern for reducing latency. These requests, which batch together access to several data elements, typically generate tens to thousands of operations performed at backend servers, each hosting a partition of a large dataset. This makes reducing latency at the tail even more imperative because the larger a fan-out request is, the more likely it is to be affected by the inherent variability in the latency distribution, where the 99th-ile latency can be more than an order of magnitude higher than the median.

The latency percentile has low, middle, and **tail** parts. Controlling latency is essential and the techniques we can utilize to achieve that are quite applicable for scale-out distributed systems and summarized below:

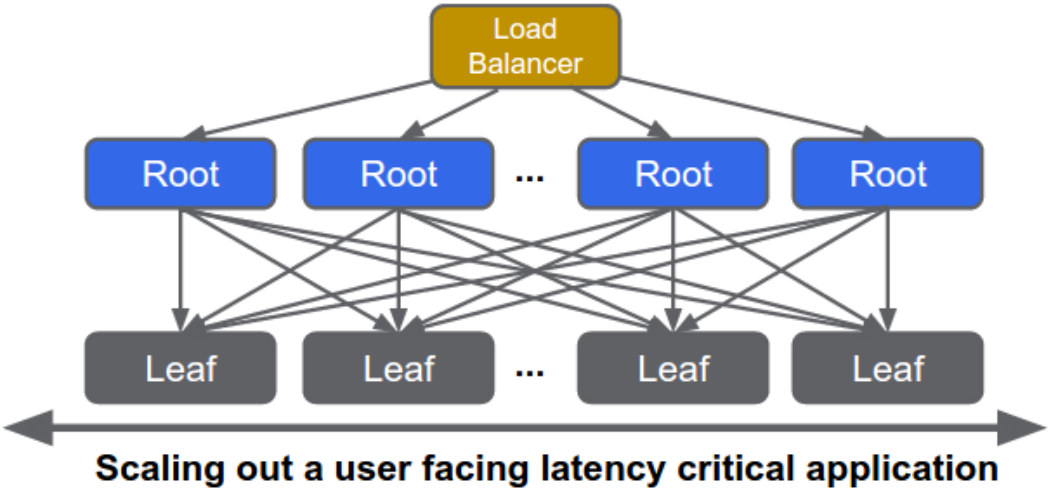
- **Low-Middle Latency Parts:**
  - **a.** Provisioning more resources
  - **b.** Cut and parallelize the tasks
  - **c.** Eliminate head-of-line blocking
  - **d.** Caching
- **Tail Latency Part:**
  The basic idea is hedging. Even weve parallelized the service, the slowest instance will determine when our request is done. You can use probability math to model the combined latency distribution.
  - **a.** Send more requests than necessary and only collect the fastest returned, helps reduce the tail. Send 2 instead of 1. Send 11 instead of 10 (e.g. in erasure-coding 10 fragment reconstruct read). Send backup requests at 95
  - **b.** Canary request, i.e. send normal requests but fallback to sending hedged requests if the canary didt finish in reasonable time.
  - **c.** Smaller task partitions (micro-partition) will help achieve smoother latency distribution percentiles.
  - **d.** Reducing head-of-line blocking. A small number of expensive queries may add up latencies to a large number of concurrent cheaper queries. Uniformly smaller tasks partitioning can help.

[1] [2] [3]

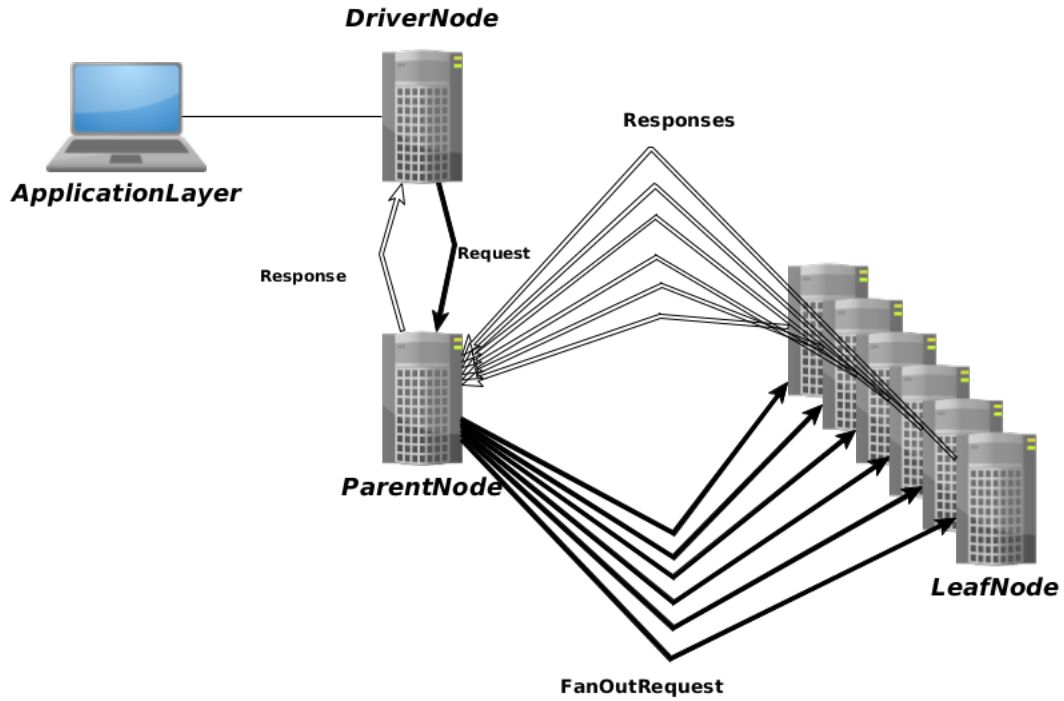
Background

Oldisim is a framework to support benchmarks that emulate Online Data- Intensive (OLDI) workloads. It models the distributed, fan-out nature of many modern applications with tight tail latency requirements, such as Google Search and some NoSQL database applications. OLDSIM is used to measure the impact of both hardware and software improvements on our scale out workloads and analyze their scaling efficiency. OLDI workloads are user-facing workloads that mine massive datasets across many servers.The benchmark emulates the fanout and request time distribution for web search. It models an example tree-based search topology. A user query is first processed by a front-end server, and eventually fanned out to a set of leaf nodes. Its main features are:

1. Strict Service Level Objectives (SLO): e.g. 99%-ile tail latency is 5ms
2. High fan-out with large distributed state
3. Extremely challenging to perform power management



Oldisim framework consists of four main entities. The search benchmark consists of four modules - ParentNode, LeafNode, All of these entities combined are responsible to emulate a real life scenario of a querying a data center. However in our approach LoadBalancer entity is not needed since our experiments focus on one RootNode (aka. ParentNode). Thus the layout is focusing on one branch of the above schema.



The DriverNode can be intuitively mapped to the terminal node which accepts queries. The ParentNode communicates with the DriverNode, which generates the requests, and is responsible for the Fanout of the request to the different terminal nodes (LeafNodes) of the data center where the data is held. The LeafNodes represent the aforementioned nodes, which belong to the data center grid by holding these data and communicate with the ParentNode. The framework emulates this process by creating requests in the DriverNode, that also holds the time statistics and pipe-lines them to the ParentNode. The ParentNode, which encapsulates a FanOutManager module distributes the requests to the LeafNodes and wait for them to finish the processing of the requests and receive a response from them that carries also the elapsed-processing time information. The processing of requests in the LeafNodes,since Oldisim works as a simulation tool, comprises from a NOP spins of the nodes' CPU. The ParentNode accordingly sends the responses to the DriverNode, where the statistics for the latency are calculated and given back to the user. Each entities operation is briefly summarized below:

DriverNode
1. Initialize the time statistics, histograms for calculating latency
2. Create virtual requests and sends them to the ParentNode
3. Waits for responses from the ParentNode to calculate the latency

ParentNode
1. Intermediate node that accepts the requests from the DriverNode
2. Encapsulates the FanoutManager module, responsible for distributing the requests to the ChildNodes
3. Waits for responses from the ChildNodes gathers them and send them back to the DriverNode

ChildNode
1. Terminal node that accepts the requestd from the ParentNode.
2. Virtually processing the requests just by spinning (no-operation) for random amount of time

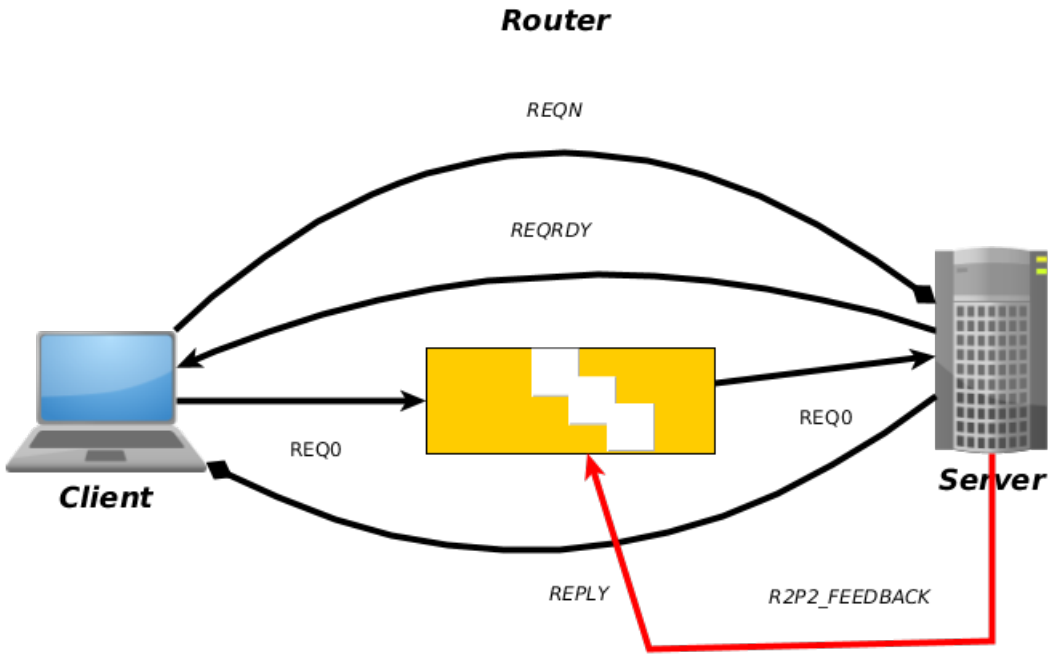
3. Create the response to the request that processed and send response to the ParentNode

The whole framework has been designed on top of libevent library and TCP socket programming. The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also support callbacks due to signals or regular timeouts. Libevent is meant to replace the event loop found in event driven network servers. In this way the communication between the entities is event based. When a step is completed the corresponding event is triggered and the corresponding actions are taken - sending requests and responses.

R2P2

We propose R2P2 (Request-Response Pair Protocol), a UDP-based transport protocol specifically targeting latency-critical RPCs within a distributed infrastructure, i.e., within a datacenter. R2P2 exposes the RPC abstraction to the network, thus allowing for efficient in-network request level load balancing. R2P2 is a connectionless transport protocol capable of sup- porting higher-level protocols such as http without protocol-level modifications. Unlike traditional multiplexing of the RPC onto a reliable byte-oriented connection, R2P2 is an inherently request/reply-oriented protocol that requires no state across requests. The R2P2 request-response pair is initiated by the client and is uniquely identified by a triplet of $< src\_IP, src\_port, req\_id >$. This design choice decouples the request destination (set by the client) from the actual server that will reply, thus enabling the implementation of any request-level load balancing policy.

describes the interactions and the packets exchanged in sending and receiving an RPC within a distributed infrastructure that uses a request router to load balance requests across the servers. We illustrate the general case of a multi-packet request and a multi-packet response.



The whole API of R2P2 can be summarized in the table below.

Application Calls		Callbacks	
r2p2_poll	Poll for incoming req/resp	**req_sucess**	Request was successful
r2p2_send_req	Send a request	**req_timeout**	Timer expired
r2p2_send_response	Send a response	**req_error**	Error condition

We note a few obvious consequences and benefits of the design:

i. Given that an RPC is identified by the triplet, responses can arrive from a different source than the original destination, thus responses are sent directly to the client, bypassing the router

ii. There is no head-of-line blocking resulting from multiplexing RPCs on a socket, since there are no sockets and each request-response pair is treated indepen- dently

iii. There are no ordering guarantees across RPCs

iv. The protocol is suited for both short and long RPCs; by avoiding the router for REQn message and replies, the router capacity is only limited by its hardware packet processing rate, not by the overall amount of bandwidth of the messages. R2P2 follows the end-to-end argument in systems de- sign

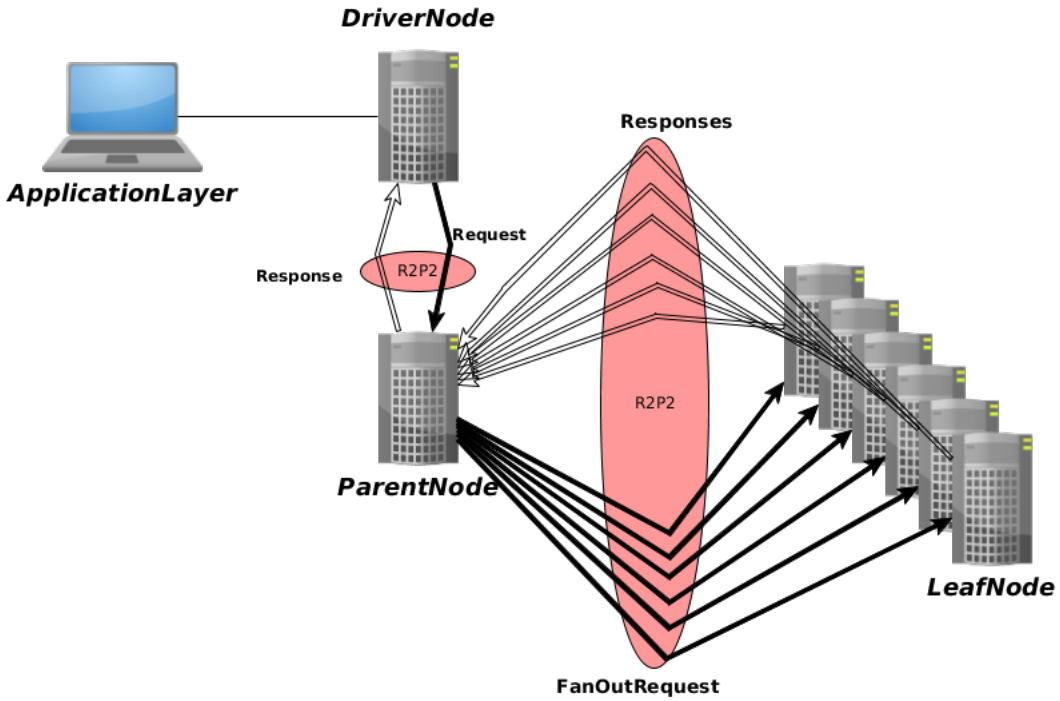
A client application initiates a request-response pair and determines the failure policy of each RPC according to its specific needs and SLOs. By propagating failures to the application, it is free to choose between at-least-once and at-most-once semantics by re-issuing the same request that failed.

Advantages compared to TCP

Unlike TCP, failures affect only the RPC in question, not other requests. This is useful in cases with fan-out replicated requests. Unlike protocols that blindly provide reliable message delivery, R2P2 exposes failures and delays to the application, thus providing system support for the implementation of tail-mitigation techniques, such as hedged requests.

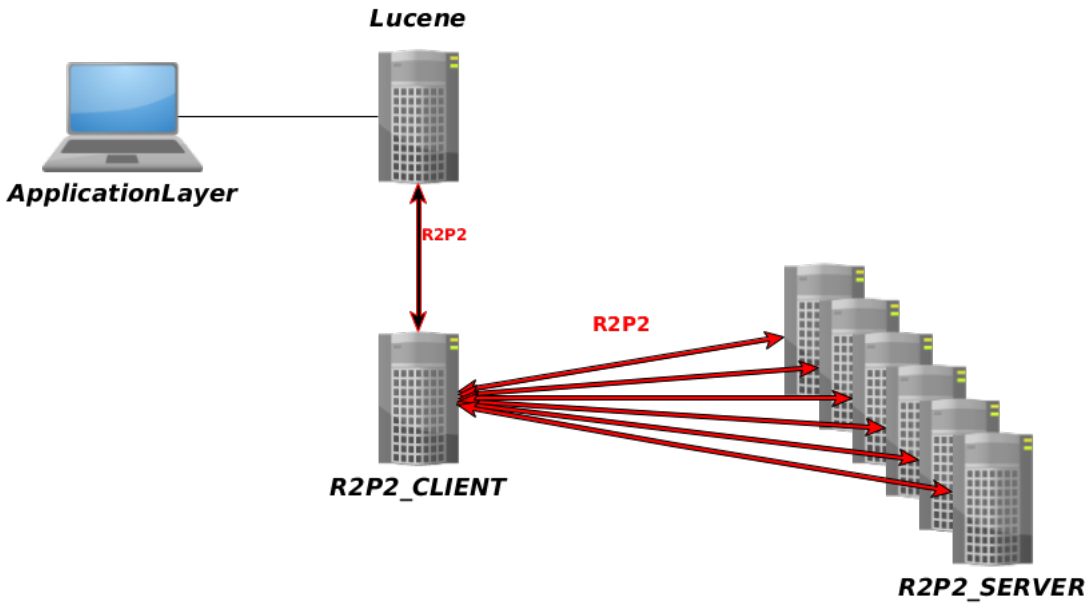
Investigation/Design

Due the resemblance of the R2P2 functionality with the combination of libevent and sockets that Oldisim uses - but based on different mechanisms and design choices- our initial approach was to substitute the event base communication (libevent) and the communication layer of the framework with our own R2P2 framework. The main motivation for this choice is that this way first we would built a tool which we would be able to utilize in the future for our own measurements based on a our own framework. Secondly, we were eager to know due to the differnces of the two approaches mainly on the network layer, messages, connection side how the R2P2 Framework will behave. Would it give us better tail latency measurements than the libevent combined with the sockets over TCP. The idea comes from the fact that in these scenarios and simulations the bottleneck of the operations is the network and how the FanOut of the requests is performed. In a next stage we would like to try to test the code of the R2P2 framework, so it uses DPDK instead of UDP sockets and re-run the benchmark to research a further boost in the performance of the FanOut operation.



Alternative Solutions

Our initial approach was to remove the event base communication (libevent) of the framework with our own R2P2 framework. However during this process we concluded that the framework was highly dependent on the libevent and that a huge re-factoring of the code could not be avoided. We decided to follow a different approach. We decided to inverse the process of porting the R2P2 framework to Oldisim. The main reason that led us to this solution was that our initial motivation for this project was after all to make a benchmark tool for R2P2 framework, which in comparison to the OLDISIM implements a UDP communication model and not a TCP one, and find out how this affects the latency. Thus we decided to port modules of Oldisim mainly the FanoutManager Module to R2P2 and create an integrated benchmark tool for R2P2, that would be also useful for future references.



Construction/Implementation

For the implementation of this tool we extracted some modules from Oldisim and import them to our R2P2 framework. In order to emulate the many LeafNodes of the Oldisim in the R2P2 framework we created many R2P2 servers that would run the same spinning process that the Oldisim LeafNodes would do. Moreover we created the same requests,responses as well the generator for the number of random spinning loops for the LeafNodes in order to achieve identical behaviour. We imitate the job of the ParentNode in the Oldisim framework with a R2P2 client executable, where we imported the functionality of the FanoutManager from Oldisim Framework and also the Tracker Module that is responsible for gathering the responses from the ChildNodes (R2P2 servers in our case) and feed them to DriverNode which in our case is an already tool that has been built for R2P2 Lucene.

Conclusion

We were able to port in the R2P2 Framework the Oldisim functionality to support benchmarks that emulate Online Data- Intensive (OLDI) workloads. This way we were able to evaluate the performance of the R2P2 protocol of communication compared to the commonly used TCP and event-based communication. Thus we also have a tool for future reference and testing based on the changes of R2P2 Framework.

Future Work

- Expanding the tool with multi-threading and evaluating it exhaustively.
- Adding load-balancer module in order to be able to have multiple R2P2-Clients(Roots in Oldisim)
- Testing the tool on different types of problems apart from graph searches like mining.

Attachments

Private Repository in Bitbucket `https://bitbucket.org/ktsitsi/rpc`

References

[1] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[2] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[3] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 95–110, New York, NY, USA, 2017. ACM.