Gotthard: Network Support for Transaction Processing

Theo Jepsen  Leandro Pacheco de Sousa  Huynh Tu Dang
Fernando Pedone  Robert Soulé
Università della Svizzera italiana

Introduction. Network latency has a significant impact on the performance of transactional storage systems. To reduce latency, systems typically rely on a cache to service read requests closer to the client. However, caches are not effective for write-heavy workloads, which have to be processed by the storage system in order to maintain serializability.

This poster presents Gotthard, a system that implements a new technique, called optimistic abort, which reduces network latency for high-contention workloads. Gotthard leverages recent advances in network data plane programmability to execute transaction processing logic directly in network devices. A switch running Gotthard examines network traffic to observe and log transaction requests. If Gotthard suspects that a transaction is likely to be aborted at the store, it aborts the transaction early by re-writing the packet header, and routing the packets back to the client. Gotthard significantly reduces the overall latency and improves the throughput for high-contention workloads.

Background. As network hardware becomes increasingly programmable [4, 3], several recent systems have demonstrated the benefits of tighter integration between the network and distributed applications that run atop. These benefits include reduced application complexity [9], improved application performance [6, 11], better network utilization [10], and more responsive traffic engineering [8]. Gotthard leverages the programmable network substrate to improve performance for high-contention workloads.

Gotthard is written P4 [3], allowing the switch to execute transaction processing logic. P4 provides high-level abstractions for network functionality: packets are processed by a sequence of tables; tables match header fields, and perform actions that forward, drop, or modify packets. Moreover, P4 allows for stateful operations that can read and write to memory cells called registers.

System Model. We consider a distributed system composed of client processes and a server. The server provides a key-value store. Clients execute transactions locally and then submit the transaction to the store to be committed, similarly to mini-transactions [1]. When executing a transaction, the client may read values from its own local cache. Write operations are buffered until commit time. The isolation property that the system provides is one-copy serializability: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [2].

The store implements optimistic concurrency control (OCC) [7]. All read transactions are served directly by the store. To commit a write transaction, the client submits its buffered writes together with all values that it has read. The store only commits a transaction if all values in the submitted transaction are current. As a mechanism for implementing this check, the system uses a compare(k,v) operation, which returns true if the current value of data item k is v, and false otherwise. Note that the compare operation is not exposed to the users, but is simply used by the system to implement OCC. In the event of an abort, the server returns updated values, allowing the client to immediately re-execute the transaction.

Design Overview. Fig. 1 shows a basic overview of Gotthard. Client transaction requests pass through a Gotthard switch. The switch either forwards the request to the store, or aborts the transaction and responds to the client directly. In the figure, the two cases are distinguished by color and line type.
The blue, dashed-line shows the forwarding case. When the client submits a request (1), the switch examines the transaction and logs the operations in its local cache (2). It then forwards the transaction message to the store, which can commit or abort the transaction (3). The store responds to the client with the execution result. The switch logs the result of the execution (4), and forwards the response to the client. If the client learns that the transaction was aborted, it can re-try. Otherwise, the transaction is complete (5).

Gotthard adopts an aggressive strategy for aborting transactions. It proactively updates its cache with the latest value from the switch (6), and the client examines the transaction message. When logging the operations, if the switch sees that a transaction is likely to abort based on some previously seen transaction, the switch will preemptively abort the request (7), and send a response to the client. The client can then re-submit the transaction (8).

Fig. 2 shows the results. As expected, when the percentage of write requests increases, the cache becomes less effective. In fact, when the percentage of write requests is above 15%, we see almost no benefit to performance for using the read cache. This is because the client can read stale values from the cache which then cause write transactions to abort at the store. In contrast, Gotthard significantly reduces execution time with respect to simple forwarding and caching. When the workload is only at only 25% writes, Gotthard already reduces the completion time by half.

**Outlook.** It is widely recognized that the performance of OCC protocols is heavily dependent on workload contention [5]. Gotthard is designed to address this problem. Gotthard compliments prior techniques for reducing network latency, such as using a cache to service read requests [9], by focusing on write-intensive workloads. Moreover, Gotthard provides a novel application of data plane programming languages that advances the state-of-the-art in this emerging area of research.

**REFERENCES**


