Introduction

FoundationDB is a key-value storage database with transactional support. When a client submits a transaction, the corresponding data will be sent to a proxy. The proxy will go to the master for the version of the transaction. After versioning information is retrieved, potential read and write conflicts will be sent to the resolver. The resolver will determine if the conflicts can be resolved or reject the commit. In the case the resolution is found, the proxy will forward the key-value mutations to tLog servers, where the journals are stored and consumed by storage servers. In one transaction, multiple key-value pairs might be updated. In the current schema, all the pairs will be sent to one single proxy. If the volume of the key-value pairs is large, proxy processing, which is both CPU and network consuming, will be blocking and requests from other clients will be thus delayed. A straightforward solution of this would be, split the large transaction into multiple parts, and deliver the parts to different proxies. The details will be discussed in the following sections.

Design

Client

When a transaction comes in, the total size of the values will be estimated by summing up the size of the values of each key-value pair. If the total size is above a certain criteria, the client will split the transaction into parts. The number of parts is determined by the number of proxies the client connects to. If the client connects to N proxies, then the transaction will be split into N parts, and each proxy will receive one of them, respectively. Since the parts need to be merged back later, they will share a same unique ID, denoted by split ID. The content of the transaction will be distributed over the parts. For read_conflicts and write_conflicts, since the elements are unordered, round-robin arrangement is sufficient; while for mutations, items are ordered as they are not commutative. To ensure the ordering, each part will handle a continuous subversion of the mutations. For example, if there are ten mutations and three proxies, then one possible distribution will be: proxy 1 receives mutation 1, 2 and 3; proxy 2 receives mutation 4, 5 and 6 whereas proxy 3 receives all the remaining mutations.

Proxy

Each proxy will receive one part of split transaction. Even the current implementation will batch the commits it have received for the purpose of performance, a part of a split transaction will **not** be batched, i.e. it will always be treated as a single transaction. There are certain reasons for this design:

1. If a part of split transaction is batched with some other non-split transactions, while the split transaction is rejected by the resolver, then the

rejection will include unnecessary victims.
2. If a part of split transaction is batched with some other parts of split transactions, this will cause the split transactions share the same version.

Both of the issues are not unsolvable, yet unnecessary complexity would be introduced; thus it is decided that parts of the split transation will always be committed alone.

Master Server

The master server provides version for each transaction. Without split transaction, the responding version is strictly monotonically increasing. However, for a split transaction, all parts should use the same prevVersion and version, to ensure they are processed at the same time. To achieve this, when the master server receives a version request from a split transaction for the first time, the split ID and version will be stored in a cache. The rest of the parts, identified by the split ID, will re-use the version in the cache. Each split ID/version pair in the cache will have a timestamp. The timestamp is used to determine if the pair is expiring or not. The same approach is used to ensure prevVersion is consistent for all parts of a split transaction.

Resolver

The resolvers need to retrieve all read_conflicts and write_conflicts for a split transaction to carry out its task. Currently, this is done by waiting for all proxies sending the conflicts and merging them later. An alternative approach is to let one part, denoted by **P**, carry all the conflicts, and cache the result of the resolution per split ID. Parts arrived before **P** will need to wait; while parts arrived after **P** can immediately receive the result from the cache. The benefit is that the resolvers do not need to wait for all parts; while the bandwidth might be the bottleneck since only one proxy retrieves and delivers the conflict range.

Transaction Log

Similar to the resolvers, the transaction log servers has to retrieve all assigned mutations before it commits the transaction. The mutations are sorted by subversion, as TLogs depends on the subversion of the mutations during the persistent process.