Diego: A Flexible Conflict Resolution Framework

Predrag Gruevski Kiril Vidimče David Xiao
Massachusetts Institute of Technology

Abstract

We present Diego, a simple but flexible and expressive conflict-resolution framework. Diego allows developers to easily build multi-user applications that operate on the same data in a consistent fashion. Diego resolves conflicts via developer-provided and domain-specific conflict resolution methods. Diego also provides all-or-nothing atomicity, at-most-once execution and durability. We demonstrate the framework on three different applications: a simple key-value store, a collaborative text editor and a distributed lego design editor.

1. Introduction

With the rise of increasingly complex mobile and web applications, users have come to expect that their applications run across multiple platforms, allow multiple users to manipulate the same data concurrently and consistently and remain responsive even in the presence of network issues such as dropped or delayed data. These applications often have relatively little data -- usually a few megabytes, almost certainly no more than a few hundred megabytes; however, the operations executed on this data tend to be unusual and highly domain-specific, and sometimes very complex. Designing a system that can satisfy these requirements while also scaling to thousands or millions of users is a challenge that only a handful of companies have tackled.

In this paper we present Diego, a flexible framework that we developed to solve exactly the problem described above. Diego is designed to allow rapid development of collaborative applications by supporting the execution of arbitrary, developer-specified transactions on in-memory state in an unconstrained developer-specified format. Since clients may attempt concurrent transactions, these transactions may be conflicting; Diego allows the developer to easily specify powerful conflict resolution policies that can modify conflicting transactions to avoid the conflict, overwrite the conflicting data or reject transactions altogether. Assuming correct use of the framework, Diego's transactions also offer all-or-nothing atomicity, at-most-once execution, strict serializability, as well as optional durability at some performance cost.

2. Related Work

Many modern mobile and web-based services implement a form of conflict-resolution in the presence of multiple users collaborating with the same data. High-profile services include Apple's iCloud (formerly, MobileMe) that provides transaction-based synchronization of user contact and calendar information. This transaction-based approach was first described in Bayou, a weakly-connected replicated storage system [Terry 1995]. More recently, Google developed a collaborative document preparation and editing platform (Google Docs) and the

short-lived, but high-profile collaboration platform called Google Wave (now Apache Wave). Both of these services represent examples of applications implemented using operational transformations, first described in the work by Ellis [Ellis 1989]. Operational transformations are a form of consistency control and conflict resolution by transforming each operation into one that takes into account any operations that may have occurred concurrently. An alternative approach is using differential synchronization which attempts to describe each operation as a sequence of difference and patch operations [Fraser 2009]. Diego has a more general conflict resolution model than either of these approaches, as it allows both operational transforms and differential synchronization to be expressed in terms of Diego operations.

3. Design Principles and Goals

Diego is inspired by ideas employed in functional programming and it relies heavily on the deterministic execution of abstract transactions on abstract in-memory state. Since transactions are deterministic and contain no side effects, an application server that uses Diego (from now on called the *Diego server*) and all of its clients can agree on a common state if they agree on what the *nil* (empty) state is, and on the ordered set of non-conflicting transactions that have executed. The role of the Diego server is to maintain this ordered set of transactions and resolve any transaction conflicts.

We will discuss conflict resolution by first observing that a client that has observed the entire ordered set of transactions cannot generate a conflicting transaction -- the client and server agree on the same state, so any newly generated transaction by the client are valid by definition. Assuming that the server informs clients of transactions in their execution order (see Section 4), we then only consider a transaction T to be *in conflict* if the client C that generated it was oblivious of transactions T_1, T_2, …, T_k which the server S had already executed. We call the set of such transactions T_1 through T_k the *conflict window*. It is possible that the transactions in the conflict window are independent of T, and T should proceed unchanged; this is a *false conflict* -- an example is two independent writes to different keys in a key-value store. It is also possible that C should never have attempted transaction T, and it should not be executed by the server; this is a *true conflict* -- an example of such a situation is modification of a data item, when that data item has been deleted in the conflict window. It is also possible to have a true conflict where had C been aware of T_1, …, T_k, it would have constructed a different transaction T' to achieve its desired effect. If S knows the conflict window and the state as observed by the client, S may then use that knowledge to transform T into T' and execute the transaction; in this case, the conflict is *resolvable*, and this is the idea of operational transforms. Diego allows the developer to easily and succinctly express the logic to determine which of these scenarios is occurring, and how to resolve the conflict.

Diego does not have a data model of its own, and does not constrain transactions in any way. Instead, it requires the developer to implement the data structures that represent the application state and the functions through which Diego will apply transactions to the state and resolve conflicts.

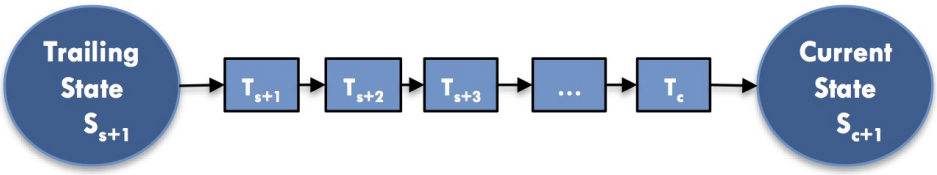

Figure 1. Trailing state, transaction log and current state

4. Framework Details

Diego achieves the goals outlined in Section 3 by maintaining two copies of the in-memory state: the *current state* copy reflecting all transactions executed up to the present moment, and the *trailing state* copy, which reflects all but the *t* most recent transactions (see Figure 1). The variable t is set by the developer; we will refer to it as the *trailing distance*. A large trailing distance allows for the resolution of transactions with longer conflict windows. However, it also linearly increases Diego's memory footprint, so the developer can tune this parameter for optimal performance in their application. Trailing distances of up to a few thousand transactions still give reasonable performance, as discussed in Section 6.

All transactions that have been applied on the *current state* but have not been applied on the *trailing state* are entered into a transaction log (Figure 1), implemented as a doubly-linked list. In the steady state, the log contains *t* transactions; Diego then executes non-conflicting transactions by applying them on the current state, appending the transaction to the head of the log, removing the transaction at the tail of the log and applying it to the trailing state. In case of a conflicting transaction, Diego first attempts to resolve the conflict (Section 4.1), and if successful, runs the above procedure on the transaction produced by the resolution process.

In Diego, transactions and states are assigned numerical IDs starting from 0. A transaction ID *i* implies that this transaction executes after the transaction with ID *i - 1*, while a state ID of *j* implies that the last transaction that was applied to this state had an ID of *j - 1*. Therefore, when a client is generating a transaction, that transaction bears the same ID as the state ID on that client.

We will discuss three scenarios regarding incoming transactions that need processing by the Diego framework:

(i) Incoming transaction ID = current state ID.

> In this case, the client who sent the transaction is up-to-date, and the transaction cannot be in conflict. Diego will simply execute the transaction as described above. This is the case with transaction T_A in Figure 2.

(ii) Incoming transaction ID < trailing state ID.

The client that sent the transaction is too far behind, and Diego cannot apply this transaction because it may violate at-most-once execution as described in Section 4.5. This is the case with transaction T_D on Figure 2. The client is then advised to fully synchronize its state with the server.

(iii) Trailing state ID ≤ incoming transaction ID < current state ID

In this case, the incoming transaction's conflict window is a subset of the transaction log. Diego uses the conflict resolution process described in the following section to determine the correct way to proceed. In Figure 2, transaction T_B was sent by a client who last observed transaction T_{s+2}, and T_C was sent by a client who last observed transaction T_{s+1}.

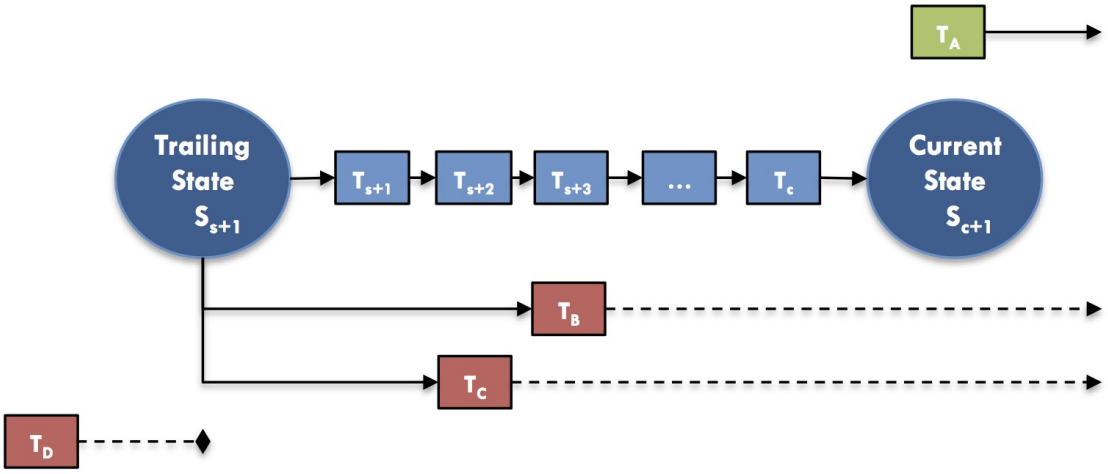

Figure 2. T_A is an example transaction that was generated with the client aware of the current state. Thus, T_A can be directly applied. In contrast, T_B and T_C were made against an older state and need to be resolved via the conflict resolution API. T_D is a transaction based on a state older than the trailing state and as such cannot be resolved automatically. In this case, the client is advised to fully synchronize its state.

4.1 Conflict Resolution API

Diego requires the developer to implement three functions, in addition to functions that check and set transaction and state IDs:

MakeState() → state

Apply(state, transaction) → outcome, transaction

Resolve(state, transaction_log, transaction) → outcome, transaction

Diego will call *MakeState* when initializing in order to create the nil (empty) state object that the current and trailing state are initially set to.

Apply attempts to modify the state by applying the specified transaction and returns a boolean result indicating success or failure and the transaction that was applied, if any. The returned transaction may be different than the one passed in, as we will discuss in a moment. *Apply* must always succeed if the transaction and the state have the same ID. Note that by definition, any calls to *Apply* with the trailing state will satisfy this condition. If the IDs are not the same, *Apply* must have been called with the current state, and should decide the type of conflict:

(i) False conflict: *Apply* returns success, and applies and returns the transaction unchanged.

(ii) True conflict: a conflict that is resolvable knowing only the current state. *Apply* uses the state to resolve the conflict by constructing a new transaction, and applying and returning it in a successful outcome. An example of this would be a transaction that appends a value to a string. Even though other append transactions may have existed in the conflict window, the append is fully defined in terms of the current state and the value to be appended.

(iii) True conflict: a conflict which is not resolvable knowing only the current state. *Appl*y cannot resolve this conflict and returns a failure outcome. In this case, *Apply* must not have modified the state it was called with.

Resolve is called when *Apply* returns a failure outcome. The trailing state, full transaction log and the conflicting transaction are provided as arguments to *Resolve*. It then attempts to use its knowledge of the trailing state and all transactions since then (a subset of which is the conflict window) to transform the conflicting transaction into a non-conflicting one. If it succeeds, it returns a success outcome and the non-conflicting version of the conflicting transaction; the non-conflicting version of the transaction is then passed to *Apply* for application onto the current state. If *Resolve* returns a failure outcome, conflict resolution has failed and the client's transaction is permanently rejected. *Resolve* never modifies the trailing state or transaction log.

The implementations of these three functions do not need to be thread-safe, as Diego guarantees that they will not be called concurrently.

4.2 Helper API

While implementing showcase applications that use the Diego framework, we have discovered a common pattern that is repeatedly used across each application. As a result, the Helper API was born. The Helper API allows developers to implement all but the most demanding applications. Unless the application needs a highly sophisticated conflict resolution strategy, most of them can be easily implemented using the higher-level helper API. This API is composed of six main functions:

Execute(state, transaction) → transaction

CheckedApply(state, transaction) → outcome, transaction

MakeContext(state) → context

UpdateContext(transaction, context) → outcome

CommutesWith(incoming_transaction, committed_transaction, context) → bool

ResolvesWith(incoming_transaction, committed_transaction, context) → bool, transaction

CheckedApply and *Execute* operate similarly to Apply in the conflict resolution API: *CheckedApply* checks whether a transaction is in conflict with the state (calling *Execute* if not), while *Execute* applies the transaction onto the state, returning the resulting transaction as in *Apply*.

The other four functions are used in the *Resolve* step of the conflict resolution API. *MakeContext* allows the developer to extract any data from the trailing state that may be useful when resolving conflicts. Since the trailing state ID may be smaller than the incoming transaction ID, the client sending the transaction has already accounted for some of the transactions in the transaction log; *UpdateContext* is used to update the extracted context with the transactions the client had already received based on the ID of the incoming transaction. *UpdateContext* is allowed to return a failure outcome in case updating the context for some particular transaction is deemed too expensive, too difficult to implement, or for any other reason the developer may decide to not attempt to resolve the incoming transaction; the transaction will then be permanently rejected.

CommutesWith is called for every transaction T_x that the client had not received, with the intention of determining if the incoming transaction commutes with T_x. If the transactions commute in the extracted context, their order of execution is unimportant, and no further resolution is necessary. If the transactions do not commute, then *ResolvesWith* is called on the same transactions and context, and it attempts to transform the incoming transaction into an equivalent one, taking account the context and the non-commutative transaction T_x. This process is repeated for every transaction the client had not received, modifying the incoming transaction as *ResolvesWith* indicates, until the end of the transaction log is reached. If all

conflicts up to that point were resolved successfully, the transaction is accepted and executed as usual.

All arguments to these functions, except the state argument to *Execute*, must not be modified inside the functions.

4.3 Example Usage of the Helper API

A simple key-value store implementation using the Helper API is shown in Listing 1. It demonstrates that a minimal Diego server can be built in less than 100 lines of code. The key-value store treats all operations as commutative as long as they operate on different keys. The developer simply implements the basic data structures for the key-value store and the API callbacks; all other details of state maintenance and thread-safe transaction processing are handled by Diego.

4.4 All-Or-Nothing Atomicity

Assuming correct implementation of the required callbacks for either API, Diego provides all-or-nothing atomicity for transactions. Namely, in the conflict resolution API, *Resolve* must never modify any state, while *Apply* only modifies the state if the transaction is accepted, and is prohibited from modifying it otherwise. In the Helper API, only *Execute* is allowed to modify state, and it is only called if the transaction is going to succeed as determined by CheckedApply.

4.5 At-Most-Once Semantics

All transactions in Diego carry a unique token that helps Diego identify if a transaction has already been executed. The token is composed of two parts: a 64-bit client identifier which should be randomly generated by each client, and a sequentially-increasing request counter, to distinguish between different requests by the same client. Diego keeps track of all tokens for all transactions currently stored in the transaction log, which helps ensure that any transaction executed at most *trailing distance* transactions ago is not executed again. Any transactions that may have been executed more than *trailing distance* transactions ago have IDs that are smaller than the trailing state's ID, and Diego will always reject them to ensure at-most-once execution.

```
// Key value store, set key/value op and transaction (list of ops)
type KeyValueStore struct {
  data map[string]string
  id int64
  resolveFn func(ancestorState *types.State, log *list.List,
                 current types.Transaction) (bool, types.Transaction)
}

// Set-value-for-key transaction
type KeyValueXa struct {
  Xid int64
  Key string
  Value string
}

func (xa *KeyValueXa) Id() int64 {
  return xa.Xid
}

func (xa *KeyValueXa) SetId(id int64) {
  xa.Xid = id
}

// Implement basic interface:
// 1. Execute, i.e., apply transaction
// 2. Apply transaction as long as it is made against current state
// 3. No context necessary for conflict resolution, so, trivial context callbacks
// 4. Transactions commute as long as they don't modify the same key
// 5. If there is a conflict, it cannot be resolved.
func (xa *KeyValueXa) Execute(s types.State) types.Transaction {
  s.(*KeyValueStore).data[xa.Key] = xa.Value
  return xa
}

func (xa *KeyValueXa) CheckedApply(s types.State) (bool, types.Transaction) {
  success, newXa := helpers.ApplyIfUpToDate(s, xa, helpers.applierForExecutable)
  return success, newXa
}

func (xa *SetValueXa) MakeContext(ancestor types.State) interface{} {
  return nil
}

func (xa *SetValueXa) UpdateContext(existing types.Transaction, context interface{}) bool {
  return true
}

func (xa *SetValueXa) CommutesWith(other types.Transaction, context interface{}) bool {
  otherXa := other.(*KeyValueXa)
  return xa.Key != otherXa.Key
}

func (op *KeyValueXa) ResolvesWith(t types.Transaction, context interface{})
                                   (bool, types.Transaction) {
  // never resolves non-commutative operations
  return false, nil
}

// Create key value store with helpers
func makeState() types.State {
  result := new(KeyValueStore)
  result.id = 0
  result.data = make(map[string]string)
  result.resolveFn = helpers.CreateManagedResolver(helpers.makeContextForExecutable,
                                                   helpers.updateContextForExecutable,
                                                   helpers.commutesWithForExecutable,
                                                   helpers.resolvesWithForExecutable)
  return result
}
```

Listing 1. A simple implementation of a key-value store using the Diego framework.

4.6 Durability

As an optional feature, Diego can durably log all executed transactions to aid fault recovery. Since Diego's current and trailing state, and the transaction log are kept in memory, it is sufficient for Diego to durably write a transaction to disk after adding it to the transaction log, but before notifying any client that the transaction committed. Diego implements this functionality by automatically serializing transactions in Go's gob format, writing them to disk and then using the *fsync* system call to flush the system's buffers.

4.7 Namespaces

All functionality described previously represents a single Diego namespace. The Diego framework supports the concurrent use of multiple dynamically generated Diego namespaces, and automatically handles the necessary synchronization to ensure thread safety. Cross-namespace transactions are not supported, and as a result, transactions that target different namespaces will likely be executed in parallel for maximum performance.

5. Applications

We used the Diego framework to implement several applications: a key-value store (with substantially more involved semantics than the one shown in Listing 1), a collaborative text editor and a distributed lego design tool. All of them achieve conflict-resolution by using the Diego conflict-resolution framework. They also exhibit all-or-nothing atomicity, at-most-once semantics, and can be durably logged. The applications demonstrate three levels of complexities regarding conflict resolution: 1) the key-value store operates entirely using the Diego-provided Helper API; 2) the collaborative text editor implements operational transformations to resolve conflicts within the Diego Conflict Resolution API and 3) the lego design editor takes advantage of the generality of the framework to implement entirely domain-specific conflict resolution.

We describe each application and its usage of Diego in the following subsections.

5.1 Key-Value Store

We implemented several different operations in the key value store to help test and benchmark Diego. One of its operations is a last-write-wins set operation that always executes against the current state regardless of conflicts -- this is useful to simulate clients that are always up-to-date. Another operation is an test-and-set operation which scans the transaction log if it does not have the ID of the current state -- a long series of such operations is the worst possible case Diego can face, and is useful for benchmarking. We also implemented several other operations to stress different parts of the Diego system and wrote deterministic and randomized tests to eliminate any bugs in our implementation.

5.2 Collaborative Text Editing

We implemented the back-end for a simple collaborative text editor by building text operational transformations in Diego. We ported a well-known open-source implementation of composable, non-invertible text operation transformations [Gentle 2014] to Go and our framework. We then verified that our implementation is functionally identical to the reference code by writing tests. Our implementation of operational transformations is only 253 lines of Go, compared to the reference implementation's 420 lines of Javascript; we believe that this difference is predominantly due to the fact that we were able to leverage the Diego framework in our implementation.

5.3 Distributed Lego Design

We developed a basic, collaborative lego design editor using a client/server architecture and the Diego framework. The server is implemented in *Go* and consists of data structures that describe an individual brick, a *universe* in which bricks are placed, a set of operations that can be performed (create brick, move brick, change brick size, change brick color and remove brick), and transactions that can group one or more operations. The server is an http daemon and listens to post queries from the client, interprets those commands and formulates response messages. A client's command can consist of a transaction for execution or a request for the current state or transaction log that can be used for client synchronization.

The client is implemented using C++ and Qt and allows generating brick operations via a simple scripting interface. The editor connects with the server on start-up and retrieves its initial state. It continues to synchronize with the server via a simple polling mechanism. Whenever conflicting operations are detected, the operation is rejected and the screen briefly flashes to inform the user. A more sophisticated user-driven conflict resolution UI remains future work. The client also allows importing of STL files that are then voxelized (or *brickified)* and transformed into individual create brick commands.

The lego design editor can encounter a number of different conflicts, each of which require different strategies when detecting and resolving the conflict. Conflicting operations can either be strictly rejected, can be accepted when operating on the same brick in a non-conflicting manner, or can be transformed when chained operations make that possible. We describe example scenarios below.

Two conflicting create operations: two operations may attempt to create bricks that overlap in the lego universe. This conflict always results in rejection.

A move brick with a conflicting create brick (and vice versa): one client may move a brick into a previously available space that is now occupied by a newly created brick. Similarly, a brick may be created into a previously available space that is now occupied by a newly moved in brick. These operations are rejected.

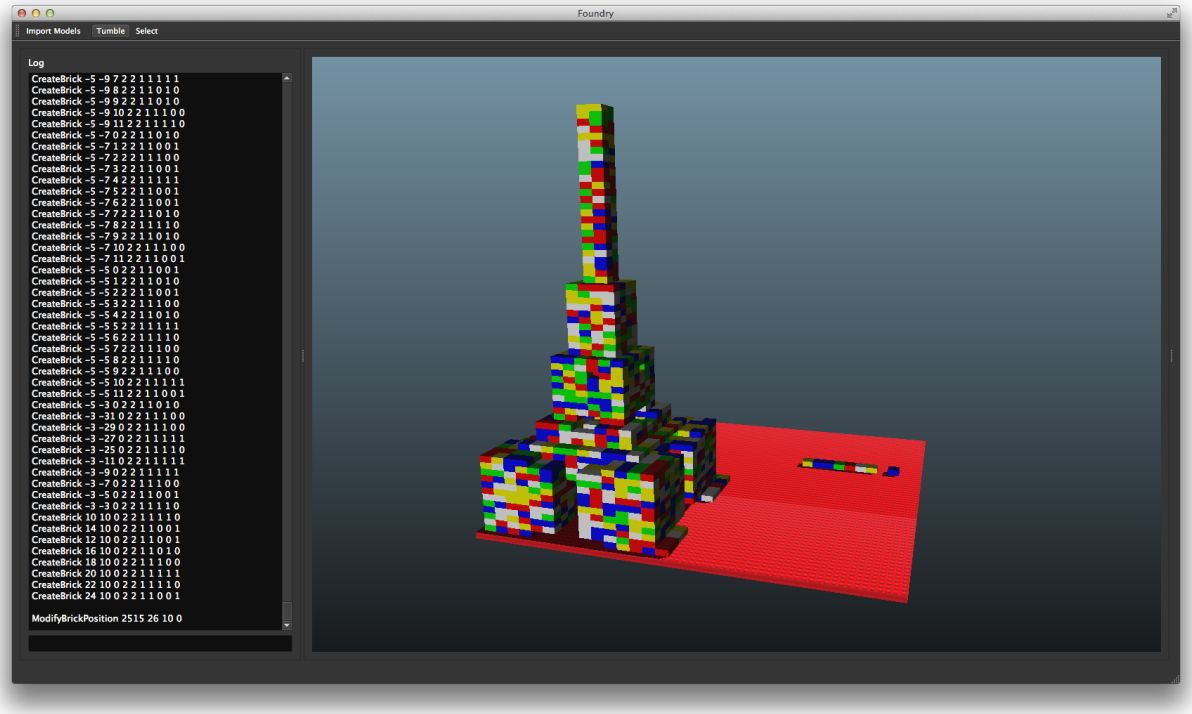

Figure 3. Example client that lets users create bricks, modify their position, size and color.

Two bricks are moved or resized in conflicting manner: one client may move or resize a brick into a space that was previously available but now occupied by another brick that was moved or resized by another client. The conflicting operation is rejected.

A brick is simultaneously modified: if two clients attempt to simultaneously modify the same aspect of the brick (position, size or color), the conflicting operation is rejected. However, if they modify different aspects of the brick (one modifies the size and the other the color), the conflicting operation is accepted.

Modifying a deleted brick: if a client attempts to modify a brick that was already deleted by another client, the operation is rejected. However, if we detect another operation that recreates an identically positioned and sized brick, we transfer the modification operation from the previously deleted brick to the newly created one. This particular example demonstrates the flexibility of our conflict resolution framework.

6. Performance

We used the key-value store implementation in Diego for our benchmarks as its transactions are lightweight and reflect Diego's efficiency more than the efficiency of the *State* data structures. We ran the benchmarks on a recent 8-core Macbook Pro equipped with an SSD.

Diego with durability disabled is highly performant, achieving over 18.45 million operations per second in the best case when all transactions are up-to-date with the state (Figure 4a). In the worst case, when all transactions require a *Resolve* call that reads the entire transaction log, performance is a function of the trailing distance (Figure 4b). Even though this case is highly unlikely in practice, as Diego returns all operations that the client has not seen after every one of their transactions, Diego still performs admirably with over 54800 transactions per second at a trailing distance of 1000 operations.

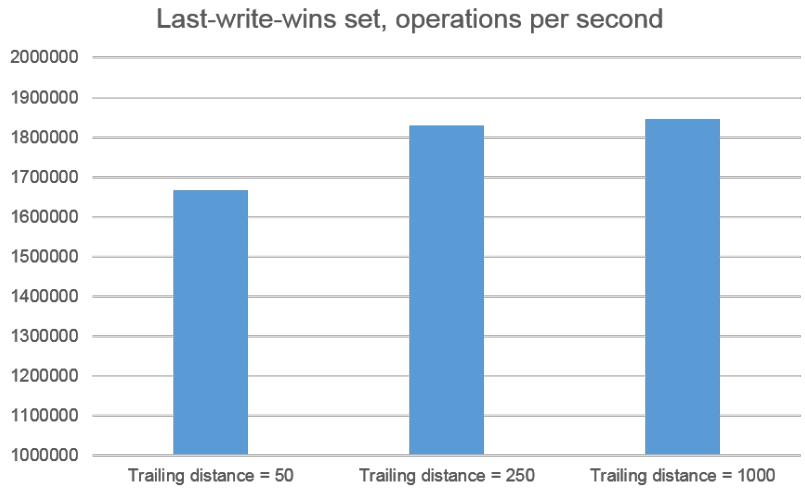

Figure 4a: Serial throughput when using the last-write-wins set operation without durability.

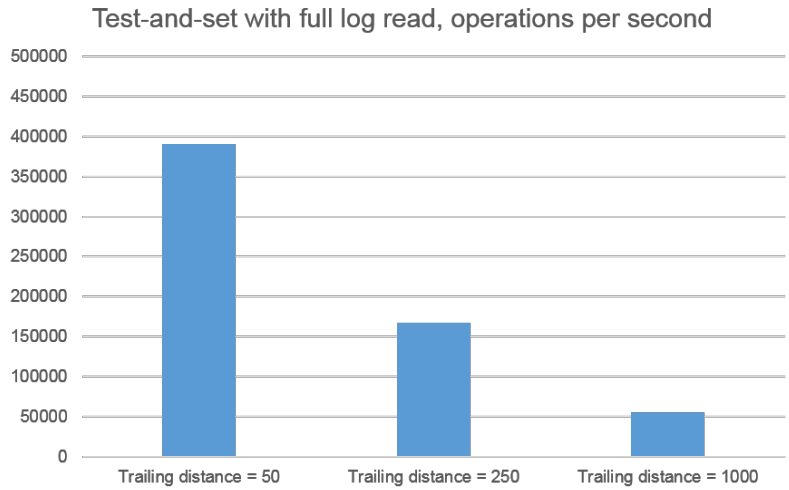

Figure 4b: Serial throughput when using the test-and-set operation without durability.

When durability is enabled, Diego takes a significant performance hit, and is IO-bound due to the fact that it has to flush the OS buffers after every transaction. As can be seen on Figure 5a, Diego peaks at 7942 transactions per second in the best case; in the same unlikely worst case as above, Diego manages 6105 transactions per second at a trailing distance of 1000 transactions.

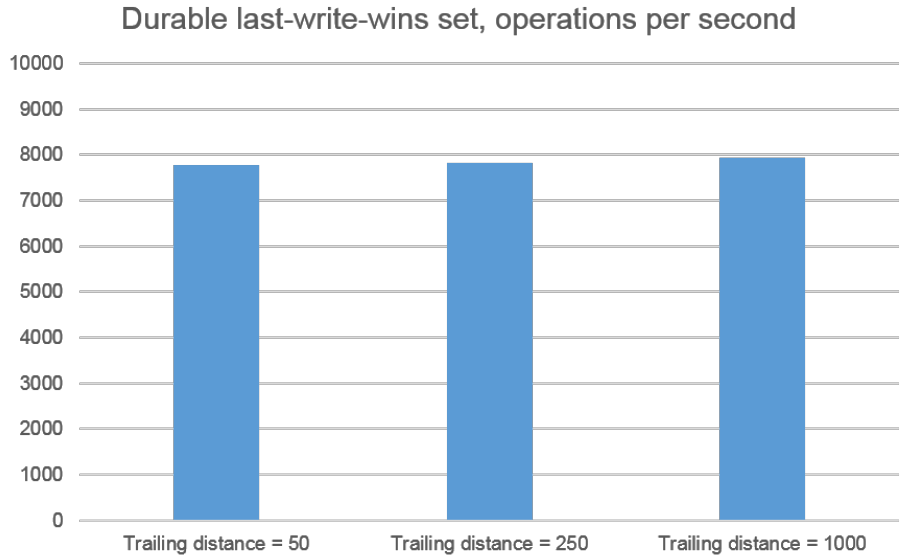

Figure 5a: Serial throughput when using the last-write-wins strategy for a key-value store with durability.

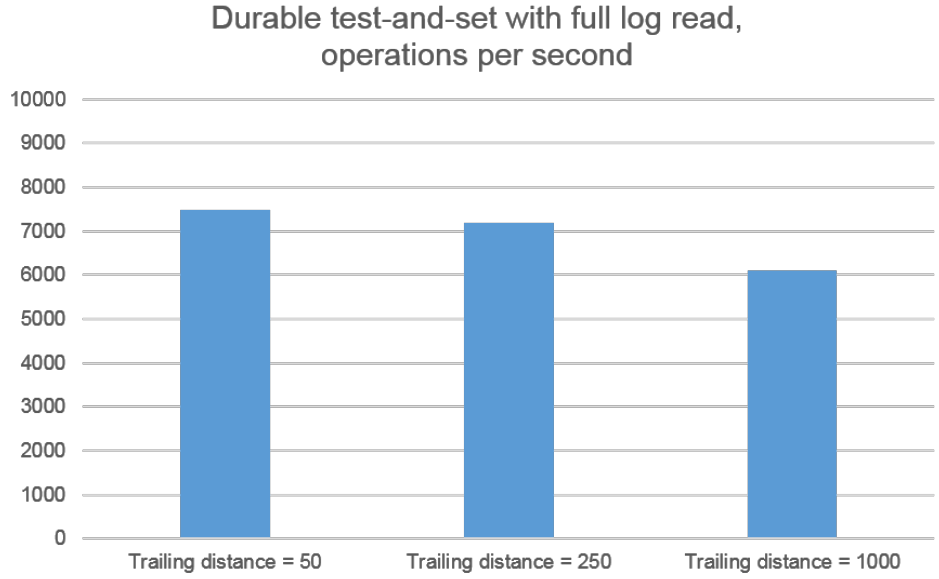

Figure 5b: Serial throughput when using the test-and-set operation for a key-value store with durability.

Diego's latency is highly sensitive to the frequency of concurrent operations and conflict resolution. Figure 6 shows the average latency as seen by clients under different namespace configurations and different numbers of concurrent clients. In three test runs, clients randomly submitted to 1, 4, or 8 namespaces. In the fourth test run provided for reference, each client submitted operations to a unique namespace independent of other clients. Results show that latency increases slightly less than linearly as contention for the same namespace increases.

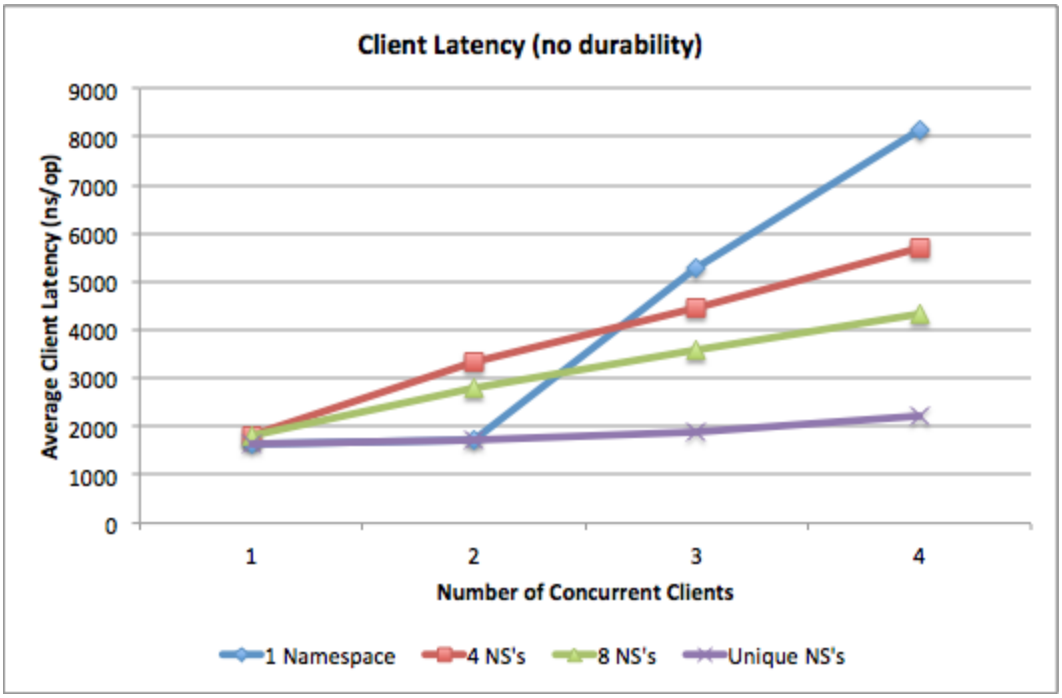

Figure 6: Client latency as the number of concurrent clients and namespace configurations changes, using a test-and-set operation against a key-value store. Durability was disabled for these tests.

References

[Ellis 1989] C. A. Ellis and S. J. Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (SIGMOD '89), James Clifford, Bruce Lindsay, and David Maier (Eds.). ACM, New York, NY, USA, 399-407.

[Fraser 2009] Neil Fraser. 2009. Differential synchronization. In *Proceedings of the 9th ACM symposium on Document engineering* (DocEng '09). ACM, New York, NY, USA, 13-20.

[Gentle 2014] Joseph Gentle, Jeremy Apthorp, Operational Transform Types. https://github.com/share/ottypes

[Terry 1995] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.* 29, 5 (December 1995), 172-182.