PLANET: Making Progress with Commit Processing in Unpredictable Environments

Gene Pang Tim Kraska♠ Michael J. Franklin Alan Fekete†

UC Berkeley
{gpang, franklin}@cs.berkeley.edu

♠Brown University
tim_kraska@brown.edu

†University of Sydney
alan.fekete@sydney.edu.au

ABSTRACT

Latency unpredictability in a database system can come from many factors, such as load spikes in the workload, inter-query interactions from consolidation, or communication costs in cloud computing or geo-replication. High variance and high latency environments make developing interactive applications difficult, because transactions may take too long to complete, or fail unexpectedly. We propose Predictive Latency-Aware NEtworked Transactions (PLANET), a new transaction programming model and underlying system support to address this issue. The model exposes the internal progress of the transaction, provides opportunities for application callbacks, and incorporates commit likelihood prediction to enable good user experience even in the presence of significant transaction delays. The mechanisms underlying PLANET can be used for admission control, thus improving overall performance in high contention situations. In this paper, we present this new transaction programming model, demonstrate its expressiveness via several use cases, and evaluate its performance using a strongly consistent geo-replicated database across five data centers.

1. INTRODUCTION

Modern database environments significantly increase the uncertainty in transaction response times and make developing user facing applications more difficult than ever before. Unexpected workload spikes [9], as well as recent trends of multi-tenancy [14, 16], and hosting databases in the cloud [22, 30, 12] are all possible causes for transaction response times to experience higher variance and latency. With modern distributed, geo-replicated databases, the situation is only worse. Geo-replication is now considered essential for many online services to tolerate entire data center outages [3, 11, 2, 1]. However, geo-replication drastically influences latency, because the network delays can be 100's of milliseconds and vary widely. Figure 1 exhibits the higher latency (~$100ms$ average latency) and variability (latency spikes exceeding $800ms$) of RPC message response times between different Amazon EC2 regions.

With these modern database environments of higher latency and variance, there are currently only two possible ways developers can deal with transactions in user facing applications such as web-shops or social web-applications. Developers are forced to either wait longer for the transaction result, or be uncertain of the transaction result. This results in an undesirable situation, as clients interacting with database-backed applications are frequently frustrated when transactions take too long to complete, or fail unexpectedly.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2588558.

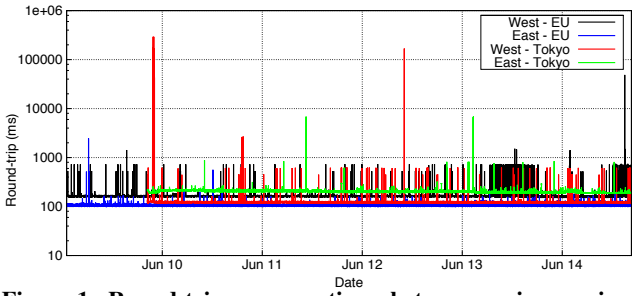

Figure 1: Round trip response times between various regions on Amazon's EC2 cluster.

To help developers cope with the uncertainty and higher latency, we introduce Predictive Latency-Aware NEtworked Transactions (PLANET), a new transaction programming model. PLANET provides staged feedback from the system for faster responses, and greater visibility of transaction state and the commit likelihood for minimizing uncertainty. This enables building user facing applications in unpredictable environments, without sacrificing the user experience in unexpected periods of high latency. Also, PLANET is the first transaction programming model that allows developers to build applications using the *guesses and apologies* paradigm as suggested by Helland and Campbell [19]. By exposing more transaction state to the developer, PLANET also enables applications to speculatively initiate some processing, thereby trading the expense of an occasional apology or revocation for faster response in the typical case that the transaction eventually succeeds. Furthermore, in cases where delays or data access patterns suggest that success is unlikely, PLANET can quickly reject transactions rather than spending user time and system resources on a doomed transaction. With the flexibility and insight that PLANET provides, application developers can regain control over their transactions, instead of losing them in unpredictable states after a timeout. The key contributions of this paper are:

- the new transaction programming model that exposes details of the transaction state, and allows callbacks
- a demonstration of how PLANET can predict transaction progress using a novel commit likelihood model for a Paxos-based geo-replicated commit protocol
- optimizations for a strongly consistent database and an evaluation of PLANET with five geographically diverse data centers

```
Session sess = factory.openSession();
Transaction tx = null;
try {
  tx = sess.beginTransaction();
  tx.setTimeout(1); // 1 second
  // The transaction operations
  boolean success = tx.commit();
} catch (RuntimeException e) {
  ...
} finally {
  sess.close();
}
```

Listing 1: Typical Hibernate Transaction

The remainder of this paper describes the details and features of PLANET. Section 2 describes how current models fall short, and outlines how PLANET satisfies our vision for a transaction programming model for unpredictable environments. In Sections 3 and 4, we describe PLANET, and in Section 5, we discuss the commit likelihood models and implementation details for a geo-replicated database. Finally, various features of the PLANET transaction programming model are evaluated in Section 6, followed by related work and conclusion in Sections 7 and 8.

2. THE PAST, THE DREAM, THE FUTURE

In an unpredictable, high variance environment, unexpected periods of high latency can cause database transactions to either take a long time to complete, or experience unusually high failure rates. In the following, we first outline the pitfalls of current techniques when dealing with high latency and variance, describe the requirements for a latency-aware transaction programming model, and finally provide an overview of our solution.

2.1 The Past: Simple Timeouts

Current, state-of-the-art transaction programming models, such as JDBC or Hibernate, provide little or no support for achieving responsive transactions. Existing transaction programming models only offer a time-out and ultimately implement a "fire-and-hope" paradigm, where once the transaction is started, the user can only hope that it will finish within the desired time frame. If the transaction does not return before the application's response-time limit, its outcome is entirely unknown. In such cases, most applications choose to display a vague error message.

Listing 1 shows a typical transaction with Hibernate [4] and the timeout set to 1 second. That is, within 1 second, the transaction either returns with the outcome stored in the Boolean variable *success*, or throws an exception. In the case of an exception, the outcome of the transaction is unknown.[1] Exceptions can either mean the transaction is already committed/aborted, will later be rolled-back because of the timeout, or maybe even was never accepted at the server and will simply be lost.

When a transaction throws an exception, developers have two options to recover from this unknown state: either periodically poll the database to check if the transaction was executed, or "hack" the database to get access to the persistent transaction log. The first option is difficult to implement without modifying the original transaction, as it is often infeasible to distinguish between an application's own changes and changes of other concurrent transactions. The second option requires detailed knowledge about the internals of the database system and is especially difficult in a distributed database system with no centralized log. PLANET provides the properties in section 2.2 to make developing applications easier, especially with geo-replicated databases.

[1]Hibernate supports *wasRolledBack()*, which returns *true* if the transaction rolled back. However, this only accounts for application-initiated rollbacks, not system rollbacks.

```
1  val t = new Tx(300ms) ({
2    INSERT INTO Orders VALUES (<customer id>);
3    INSERT INTO OrderLines
4      VALUES (<order id>, <item1 id>, <amt>);
5    UPDATE Items SET Stock = Stock - <amt>
6      WHERE ItemId = <item1 id>;
7  }).onFailure(txInfo => {
8    // Show error message
9  }).onAccept(txInfo => {
10   // Show page: Thanks for your order!
11 }).onComplete(90%)(txInfo => {
12   if (txInfo.state == COMMITTED ||
13       txInfo.state == SPEC_COMMITTED) {
14     // Show success page
15   } else { // Show order not successful page }
16 }).finallyCallback(txInfo => {
17   if (!txInfo.timedOut) // Update via AJAX
18 }).finallyCallbackRemote(txInfo => {
19   // Email user the completed status
20 })
```

Listing 2: Order purchasing transaction using PLANET

2.2 The Dream: LAGA

As described in the previous sub-section, current transaction programming models, like JDBC or Hibernate, leave the user in the dark when a timeout occurs. Furthermore, they provide no additional support for writing responsive applications. In the following we decribe the four properties a transaction programming model should have for uncertain environments, referred to as the ***LAGA*** properties for ***Liveness***, ***Assurance***, ***Guesses***, and ***Apologies***:

Liveness: The most important property is that the application guarantees liveness and does not have to wait arbitrarily long for a transaction to finish. This property is already fulfilled by means of a timeout with current transaction programming models.

Assurance: If an application decides to move ahead without waiting for the final outcome of the transaction (e.g., after the timeout), the application should have the assurance that the transaction will never be lost and that the application will at some point be informed about the final outcome of the transaction.

Guesses: The application should be able to make informed decisions based on incomplete information, before the transaction even completes to reduce perceived latency, as, for instance, suggested by Helland and Campbell [19]. For example, if a transaction is highly likely to commit or abort, an application may choose to advance instead of waiting for transaction completion.

Apologies: If a mistake was made on an earlier guess, the application should be notified of the error and the true transaction outcome, as suggested in [19], so the application can apologize to the user and/or correct the mistake.

The ***LAGA*** properties describe a transaction programming model, and are orthogonal to the transaction guarantees by the underlying database. For example, a database can fully support the ***ACID*** properties (Atomicity, Consistency, Isolation, Durability), while the transaction programming model supports the ***LAGA*** properties.

2.3 The Future: PLANET Example

In contrast to the state-of-the-art transaction programming models, PLANET fulfills not only the ***Liveness*** property but all four properties. Listing 2 shows an example transaction using PLANET in Scala. The transaction is for a simple order purchasing action in a web shop, such as Amazon.com. The code fragment outlines how the application can guarantee one of three responses to the user within 300ms: (1) an error message, (2) a "Thanks for your order" page, or (3) a successful order page, given the status of the trans-

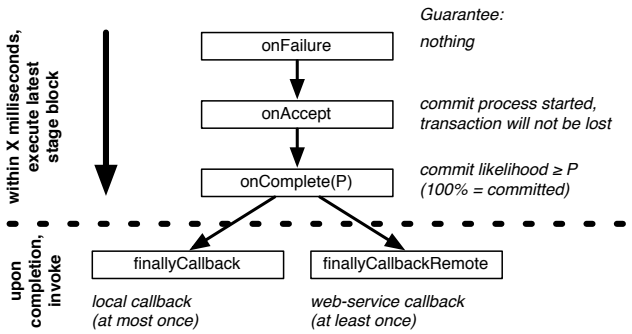

Figure 2: Client view of PLANET transactions

action at the timeout; Further, it guarantees an email and AJAX notification when the outcome of the transaction is known. In the following, we briefly explain the different mechanisms used in the example, before Section 3 describes the semantics of PLANET .

With PLANET, transaction statements are embedded in a transaction object (line 1–6). PLANET requires a timeout (line 1) to fulfill the ***Liveness*** property. After the timeout, the application regains control. PLANET exposes three transaction stages to the application, *onFailure*, *onAccept* and *onComplete*. These three stages allow the developer to appropriately react to the outcome of the transaction given its state at the timeout. Whereas the code for *onFailure* (line 7–8) is only invoked in the case of an error, and *onComplete* (line 11–13) only if the transaction outcome is known before the timeout, *onAccept* exposes a stage between failure and completion, with the promise that the transaction will not be lost, and the application will eventually be informed of the final outcome. Therefore, the *onAccept* stage satisfies the ***Assurance*** property.

Only one of the code fragments for *onFailure*, *onAccept*, or *onComplete* is executed within the timeframe of the timeout. In addition, *onComplete* can take a probability parameter that enables speculative execution with a developer-defined commit likelihood threshold (90% in the example) and thus, fulfills the ***Guesses*** property. Finally, the callbacks *finallyCallbackRemote* and *finallyCallback* support the ***Apologies*** property, by providing a mechanism for notifying the application about the final outcome of the transaction regardless of when the timeout happened.

Given the four properties, PLANET enables developers to write highly responsive applications in only a few lines of code. The next section precisely describes the semantics of PLANET.

3. PLANET SIMPLIFIED TRANSACTION PROGRAMMING MODEL

PLANET is a transaction programming model abstraction and can be used with different data models, query languages and consistency guarantees, similar to JDBC being used with SQL or XQuery, depending on database support. The key idea of PLANET is to allow developers to specify different *stage blocks* (callbacks) for the different stages of a transaction. This section describes the simplified transaction programming model of PLANET, which is essentially "syntactic sugar" for common stages and usage patterns. Section 4.1 describes the more general model, which provides the developer with full control and customization possibilities.

3.1 Timeouts & Transaction Stage Blocks

At its core, PLANET combines the idea of timeouts with the new concept of *stage blocks*. In PLANET, the timeout is always required, but can be set to infinity. Finding the right timeout is up to the developer and can be determined through user studies [31, 6]. Listing 2 shows an example with the timeout set to $300ms$.

PLANET simplified transaction programming model also defines three *stage blocks*, corresponding to the internal stages of the transaction, that follow an ordered progression of *onFailure*, then *onAccept*, then *onComplete* (see also Figure 2). When the timeout expires, the application regains the thread of control, and depending on the state of the transaction, *only* the code for the latest defined *stage block* is executed. That is, for any given transaction, only one of the three stage blocks is ever executed. In the following, we describe the *stage blocks* and their guarantees in more detail.

onFailure. PLANET tries to minimize the uncertainty when a timeout occurs, but cannot entirely prevent it. In fact, for distributed database systems, it can be shown by a reduction to the Two Generals' Problem, that it is theoretically impossible to completely avoid uncertainty of the outcome. Therefore, PLANET requires the *onFailure* stage to be defined, which is similar to an exception code block. When nothing is known about the commit progress when the timeout expires, the *onFailure* code is executed. Reaching the *onFailure* stage does not necessarily imply that the transaction will abort; instead the application may later be informed about a successfully committed transaction (see Section 3.3).

onAccept. When the transaction will not be lost anymore and will complete at some point, the transaction is considered *accepted*. Typically, this is after the system started the commit process. How strong the *not-be-lost* guarantee is depends on the implementation. For example, in a distributed database, it could mean that at least one database server successfully received and acknowledged the commit proposal message (with the assumption that all servers eventually recover all acknowledged messages).

If the timeout expires, the *onAccept* stage is executed if the system is still attempting to commit the transaction, but the final outcome (i.e., abort or commit) is still unknown. If the later *stage block*, *onComplete*, is undefined, *onAccept* will be executed immediately after the transaction is *accepted* by the system, and not wait for the timeout. This feature is particularly useful for achieving very fast response times for transactions which will not abort from conflict (non-conflicting, append-only transactions). In contrast to the *onFailure* stage, if the *onAccept* stage is invoked, the system makes two important promises: (1) The transaction will eventually complete, and (2) the application will be later be informed about the final outcome of the transaction (see Section 3.3). Therefore, the *onAccept* stage satisfies the ***Assurance*** property.

onComplete. As soon as the final outcome of the transaction is known and the timeout has not expired, *onComplete* is executed. If the timeout did expire, an earlier stage, either *onFailure* or *onAccept*, was already executed, so *onComplete* will be disregarded.

Finally, all *stage blocks* are always invoked with a transaction state summary (*txInfo* in Listing 2). The summary contains information about the current state of the transaction (*UNKNOWN*, *REJECTED*, *ACCEPTED*, *COMMITTED*, *SPEC_COMMITTED*, *ABORTED*), the time relative to the timeout (if the transaction *timed out*), and the updated commit likelihood (Section 3.2).

3.2 Speculative Commits using onComplete

PLANET provides the ***Guesses*** property by allowing developers to write applications that advance without waiting for the outcome of a transaction, if the expected likelihood of a successful commit is above some threshold P. We refer to the ability for applications to advance before the final transaction outcome and based on the commit likelihood as *speculative commits*. The developer enables speculative commits by using the optional parameter P of the *onComplete stage block*. For example, if the developer decides that a transaction should be considered as finished when the commit likelihood is at least 90%, then the developer would define the *stage block* as *onComplete(90%)* (shown in Listing 2). For a set thresh-

old P, PLANET will execute the *onComplete stage block* before the timeout, as soon as the commit likelihood of the transaction is greater than or equal to the threshold, which can greatly reduce transaction response times.

Obviously, the commit likelihood computation is dependent on the properties of the underlying system, and Sections 5.1 and 5.2 describe the model and statistics required for a geo-replicated database using the Paxos protocol. For most databases, PLANET will calculate the commit likelihood using local statistics at the beginning of the transaction. However, for some database systems, it is also possible to re-evaluate the likelihood as more information becomes available during the execution of the transaction. Possible examples are: discovering a record of a multi-record transaction that has completed, and receiving responses of previous RPCs.

Of course, speculative commits are not suited for every application, and the commit likelihoods and thresholds vary significantly from application to application. We believe, however, that speculative commits are a simpler programming construct for applications, which can already cope with eventual consistency. We envision many additional applications which can significantly profit from PLANET's speculative commits. For example, a ticket reservation system could use speculative commits to allow very fast response times, without risking significantly overselling a high-demand event like Google I/O (see also [23] and [5]). In order to guide users in picking the right threshold we envision leveraging user experience studies or automatic cost-based techniques [23].

3.3 Finally Callbacks and Apologies

It is possible that the application will not know the transaction's final outcome when the timeout expires, either because of a speculative commits, because *onComplete stage block* was not defined, or because the transaction took too long. PLANET addresses this with code blocks ***finallyCallback*** and ***finallyCallbackRemote***, which are special callbacks used to notify the application of the actual commit decision of the completed transaction. They are different from the other *stage blocks* because they are not restricted to execute within the timeout and run whenever the transaction completes. The callbacks allow the developer to apologize and correct any "wrong" doing because of speculative commits, errors, or timeouts and satisfy the ***Apologies*** property.

In contrast to the *finallyCallback* code which can contain arbitrary code, the code for *finallyCallbackRemote* can only contain web-service invocations (e.g., REST calls), which can be executed anywhere in the system without requiring the outer application context. For *finallyCallback*, the system guarantees at-most-once execution. For example, a developer might use *finallyCallback* to update the web-page dynamically using AJAX about the success of a transaction after the timeout expires. However, *finallyCallback* might never be executed, in cases of server failures. In contrast, *finallyCallbackRemote* ensures at-least-once execution as the web-service invocation can happen from any service in the system at the cost of reduced expressivity (i.e., only web-service invocations are allowed). Listing 2 shows an example of defining both callbacks, *finallyCallback* to update the application using AJAX and *finallyCallbackRemote* to send the order email. Like the *stage blocks*, *finallyCallback* and *finallyCallbackRemote* also have access to the current transaction summary, *txInfo*.

When speculative commits are used with $P < 100\%$, some transactions may experience *incorrect commits*. This occurs when the transaction commit likelihood is high enough (greater than P) and *onComplete* is invoked, but the transaction aborts at a later time. To apologize for *incorrect commits*, the final status is notified through a *finally* callback, thus satisfying the ***Apologies*** property.

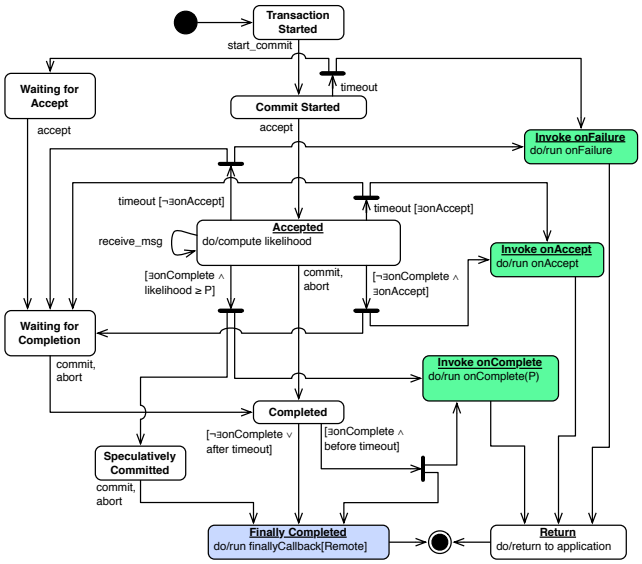

Figure 3: PLANET transaction state diagram

3.4 PLANET vs. Eventual Consistency

Utilizing the *onAccept* block, coupled with the *finally callbacks* is a valuable alternative to Eventual Consistency (EC) models [15]. EC systems are typically used for their fast response times and high availability guarantees, but come with a high cost: potential data inconsistencies. In contrast, PLANET can also offer fast response times and high availability without sacrificing data consistency.

PLANET does not change the transaction semantics of the underlying database; even with speculative commits the data would never be rendered inconsistent if the backend is strongly consistent. It only allows clients to deliberately proceed (using *onAccept* or speculative commits), even though the final decision (commit/abort) is not yet known. Furthermore, the *onAccept* stage guarantees (transactions will not be lost) can be implemented in a highly available fashion. Similar to EC, the application can still make inconsistent decisions, like informing a user about a successful order even though items are sold out. But in contrast to EC, PLANET does not allow the database to expose an inconsistent state to other subsequent transactions or concurrent clients, and thus, prevents dependencies based on inconsistent data, which are particularly hard to detect and correct. This property makes it easier to write applications than with the EC transaction model: it isolates mistakes (e.g., a commit followed by an abort) to a single client/transaction allowing the developer to better foresee the implications.

3.5 Life of a PLANET Transaction

Figure 3 formalizes the transaction programming model using a state diagram. State transitions are represented by edges and use the format "*event[guard]/action*", and the dark lines represent a fork in the state diagram, to capture parallel execution. The shaded states are the ones which execute user-defined stage blocks, and $\exists stageName$ and $\neg\exists stageName$ refer to whether or not a stage block was defined. This diagram shows that the commit likelihoods are computed every time a new message is received, and that timeouts fork execution so that the transaction continues to execute while the application regains control. For an example of a speculative commit, in the *Accepted* stage, when the *onComplete* stage block is defined and the likelihood is greater than P, the transition will fork so that one will run *onComplete* and return to the application, while the other will be in the *Speculatively Committed* state waiting for completion. Even though this state diagram may

```
val t = new Tx(30000ms) ({
  UPDATE Accounts SET Balance = Balance - 100
    WHERE AccountId = <id>;
}).onFailure(txInfo => {
  // Show error message
}).onComplete(txInfo => {
  if (txInfo.success) // Show money transfer
  else // Show failure page
}).finallyCallbackRemote(txInfo => {
  if(txInfo.success && txInfo.timedOut)
    // Inform service personnel
})
```

Listing 3: ATM example using PLANET

```
val t = new Tx(200ms) ({
  INSERT INTO Tweets VALUES (<user id>, <text>);
}).onFailure(txInfo => {
  // Show error message
}).onAccept(txInfo => {
  // Show tweet accept
})
```

Listing 4: Twitter example using PLANET

be complicated, the client sees a far simpler view, without all the internal transitions, which can be simply explained using the state progression flow shown Figure 2.

3.6 Usage Scenarios

PLANET is very flexible and can express many kinds of transactions. There are ad hoc solutions and systems for every situation, but PLANET is expressive enough to encapsulate the use cases into a single model for the developer, regardless of the underlying implementation. A document containing various use cases can be found in [5]. In addition to the web shop motivating example in Listing 2, we describe two other use cases.

ATM Banking. Code Listing 3 shows an example of an ATM transaction. The structure is very similar to a standard Hibernate transaction, where only failures and commits are reported on. This is because correctness is critical for money transfers, so waiting for the final outcome is most appropriate. Therefore, there is no *onAccept stage block*. When the timeout expires and the transaction is not completed, it is a failure. However, if the transaction times out and later commits, this means the user saw a failure message, but the transaction eventually committed successfully. The *finallyCallbackRemote* handles this problematic situation.

Twitter. Code Listing 4 shows an example of a transaction for sending an update to a micro-blogging service, like Twitter. In contrast to the ATM example, these small updates are less critical and are not required to be immediately globally visible. Also, the developer knows that there will never be any transaction conflicts, since every transaction is essentially a record append, and not an update. Therefore, the transaction only defines the *onFailure* and *onAccept* code blocks, which means the developer is not concerned with the success or failure of the commit. This type of transaction easily provides the response times of eventually consistent systems, but at the same time never allowing the data to become inconsistent.

4. ADVANCED PLANET FEATURES

The following section first describes the generalized version of PLANET, which gives developers even more freedom, and then, the PLANET admission control feature.

4.1 Generalized Transaction Programming Model

While the simplified model in Section 3 supports many of the common cases, there may be situations when the developer wants more fine-grained control. This section describes the fully generalized transaction programming model which supports the simplified model in Section 3, but also provides more control for the developer. The generalized model has only one *stage block*, *onProgress*, and has the two *finally* callbacks, *finallyCallback* and *finallyCallbackRemote*. The *stage blocks* of the simplified model in Section 3 are actually just "syntactic sugar" for common cases using the generalized model. Listing 5 is equivalent to the simplified Listing 2, but uses the generalized *onProgress* block instead.

```
val t = new Tx(300ms) ({
  // Transaction operations
}).onProgress(txInfo => {
  if (txInfo.timedOut) {
    if (txInfo.state == ACCEPTED)  // onAccept code
    else // onFailure code
    FINISH_TX  // finish the transaction.
  } else {  // not timedOut
    if (txInfo.commitLikelihood > 0.90) {
          // onComplete(90%) code
      FINISH_TX  // finish the transaction.
    }
  }
}).finallyCallback(txInfo => {
  // Callback: Update status via AJAX
}).finallyCallbackRemote(txInfo => {
  // Callback: Update status via email
})
```

Listing 5: PLANET general transaction programming model. Equivalent to Listing 2

4.1.1 onProgress

The *onProgress stage block* is useful for the application to get updates or notifications on the progress of the executing transaction. Whenever the transaction state changes, the *onProgress* block is called with the transaction summary, containing information about the transaction status and other additional information about the transaction (e.g., the commit likelihoods). This means the stage may be called multiple times during execution. By getting notifications on the transaction status, the application can make many informed decisions on how to proceed.

A special feature for the code defined in the *onProgress* block is that the developer can return a special code *FINISH_TX* to signal to the transaction handler that the application wants to stop waiting. If the code returns *FINISH_TX*, the application will get back the thread of control and get notified of the outcome with a *finally* callback. If the application does not want the thread of control, the transaction handler will continue to wait for a later progress update.

4.1.2 User-Defined Commits

Using the generalized PLANET transaction programming model and the exposed transaction state, the developer has full control and flexibility on how transactions behave. For example, *onProgress* allows informing the user details about the progress of a buying transaction;: a website could first show, "trying to contact the backend", then move on to "booking received", until it shows "order successfully completed". This is not possible in the simplified model.

The generalized model is a very powerful construct; ultimately, it allows to redefine what a commit means. For example, for one transaction, a developer can choose to emulate asynchronously replicated systems by defining the commit to be when the local data center storage nodes received the updates (assuming that the transaction summary contains the necessary information), and choose to wait for the final outcome for another transaction.

4.2 Admission Control

With commit likelihoods, it becomes very natural to also use these likelihoods for admission control. PLANET implements ad-

mission control with the hope of improving performance by preventing wasted resources or thrashing. If the system computes a transaction commit likelihood which is too low, that means there is a high chance the transaction will abort. If that is the case, it may be a better idea not to actually execute the transaction and potentially waste resources in the system such as CPU cycles, disk I/o, or extraneous RPCs. In addition to improving resource allocation, not attempting transactions with low likelihoods reduces contention on the involved records, which can lead to improving the chances for other transactions to commit on those records for some consistency protocols (e.g., MDCC [24]). Currently, PLANET supports two policies for admission control: Fixed and Dynamic.

Fixed(threshold,attempt_rate). Whenever the transaction commit likelihood is less than the *threshold*, then the transaction is attempted with probability *attempt_rate*. For example, *Fixed(40,20)* means when the commit likelihood is less than 40%, the transaction will be attempted 20% of the time. If *attempt_rate* is 100%, the policy is equivalent to using no admission control.

Dynamic(threshold). The *Dynamic* policy is similar to the *Fixed* policy, where the attempt rate is not fixed, but related to the commit likelihood. Whenever the likelihood, L, is lower than the *threshold*, the transaction is attempted with probability L. For example, a *Dynamic(50)* policy means all transactions with a likelihood, L, less than 50% will be attempted with probability L. If the *threshold* is 0, the policy is equivalent to using no admission control. Section 6.7 further investigates these parameters.

When a transaction is rejected by the system, PLANET does not actively retry the transaction. However, with PLANET, the developer may choose to retry rejected transactions (the transaction summary contains the necessary information). This may lead to starvation, but the developer can define how retries are done, and implement retries with exponential backoff to mitigate starvation.

The PLANET admission control technique using commit likelihoods does not preclude using other methods such as intermittent probing [18]. In fact, admission control of PLANET can augment existing techniques by using record access rates to improve the granularity of information, and by using commit likelihoods to identify types of transactions and access patterns.

5. GEO-REPLICATION

The PLANET transaction programming model can be implemented on any transactional database as long as the required statistics and transaction states are exposed. It is even possible to use PLANET for non-transactional key/value stores, where the commit likelihood would represent the likelihood of an update succeeding without lost updates (see also Section 5.1.3). However, we see the greatest benefits of PLANET with geo-replicated, strongly consistent, transactional database systems such as Megastore [8], Spanner [13] or MDCC [24, 20]. Figure 1 shows the long and unpredictable delays between data centers (on Amazon EC2) these systems have to deal with. In the following, we show how PLANET can be implemented on an existing geo-replicated database, MDCC, for which two implementations are publicly available [24, 20]. However, the results from this study are transferable to other systems, like Megastore or COPS[27], by adjusting the likelihood models.

5.1 Conflict Estimation for MDCC

MDCC is a distributed, geo-replicated transactional database designed along the lines of Megastore [8]. A typical MDCC database deployment is distributed across several storage nodes, and the nodes are fully replicated across multiple data centers. Besides the small changes to the underlying MDCC (see Section 5.2), most of PLANET is implemented in the client-side library.

In MDCC, every record has a master that replicates updates to remote data centers using the Paxos consensus protocol [25] similar to Megastore's Paxos implementation. However, MDCC is able to avoid Megastore's limitations of being only able to execute one transaction at a time per partition and has significantly higher throughput by using a finer grained execution strategy [24, 20]. Furthermore, MDCC supports various read-modes (snapshot-isolation, read-committed) and proposes several optimizations including a *fast* protocol to reduce the latency of commits at the cost of additional messages in the case of conflicts. For the remainder of this section, we ignore many of the optimizations and we focus on MDCC's default setting (read-committed) without the *fast* protocol. We only model the MDCC classic protocol, as this configuration is more similar to other well-known systems like Megastore.

5.1.1 The MDCC Classic Protocol

In its basic configuration, the MDCC protocol provides read-committed isolation, and ensures all write-write conflicts are detected similar to snapshot isolation, but only provides atomic durability (i.e., all or none of the updates are applied) and not atomic visibility (i.e., consistent reads). This read guarantee has proven to be very useful, because read-committed is still the default isolation level in most commercial and open-source database systems (e.g., MS SQL Server, PostgreSQL, Oracle).

In the MDCC classic protocol, the transaction manager (typically, the client) acquires an *option* per record update in a transaction using Multi-Paxos. The option is learned either as accepted or rejected using Multi-Paxos from a majority of storage nodes (note that even rejecting an option requires learning the option). A learned (but not yet visible) option prevents other updates on the same record from succeeding (i.e., does the write-write conflict detection) and can best be compared to the first phase of two-phase commit (2PC) as it prepares the nodes to commit. However, in contrast to 2PC, the transaction is immediately committed if all the options have successfully been *learned* using Paxos. If all options are learned as accepted, the transaction manager has no choice and must commit the transaction. Similarly, it has to abort the transactions if one of the options is learned as aborted. In this paper, we do

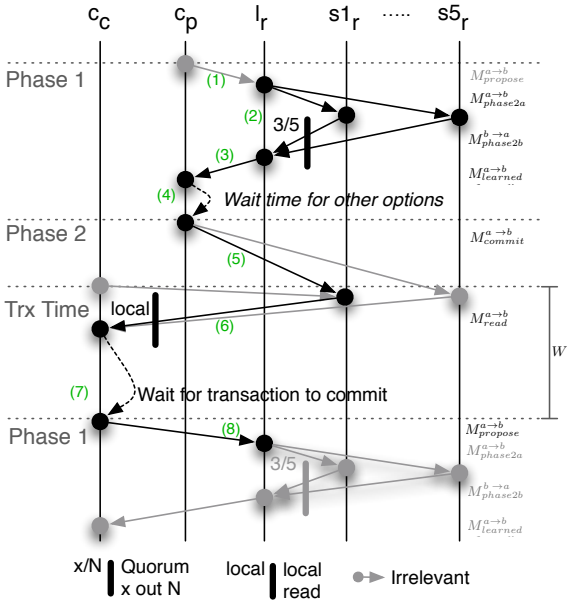

Figure 4: Time-sequence for rounds

not consider the MDCC optimization of broadcasting all messages to avoid a second phase.

The sequence diagram of the protocol is shown in Figure 4. As a first step, the client (assumed to be C_p) proposes the option to any record-leader L_r (i.e, the master responsible for learning the option) involved in the transaction shown as step (1). Afterwards, the leader executes the Paxos round by sending a Paxos *phase2a* message to all storage nodes and waits for the majority of *phase2b* answers, visualized as step (2) for 5 storage $s1_r \dots s5_r$ nodes in Figure 4. If the option is learned by a majority, the leader sends a *learned* message with the success value to the transaction manager on the client C_p, shown as step (3). The transaction manager now has to wait for all *learned* messages, one per update in the transaction, indicated as step (4).[2] If all options are learned successfully, the transaction is committed and the client is allowed to move on. However, the updates are not yet visible to other clients. To make the updates visible, the transaction manager sends a *commit visibility* message to all involved storage nodes, shown as step (5). Note that the transaction manager/client C_p, the record leader L_r, and the storage nodes $S1_r \dots S5_r$ may all be in different data centers. If the client, leader and at least one storage node are co-located in the same data center, the commit will only take a single round-trip between remote data centers (local round-trips are less significant).

5.1.2 Commit Likelihood Model for MDCC

We next show how we can model the commit likelihood for the described protocol. The key idea is estimating the time it takes to propagate the updates of a preceding transaction, so that the current transaction does not conflict with it. Given this duration, we can then calculate the likelihood of another transaction interfering by considering the update rate per record. Section 5.2 describes how we actually are able to collect the necessary statistics and precompute a lot of the calculations.

In the remainder, we assume the following symbols[3].

- $\{a, b, c, l\} \in \{1 \dots N\}$, where 1 to N are the data centers
- $C \in \{1 \dots N\}$, stochastic variable of the data center of the client, with c being an instance of the variable
- $L \in \{1 \dots N\}$, stochastic variable of the data center of the master (i.e., leader) with l being an instance of the variable
- $M^{a \to b} \in \mathbb{R}$, stochastic variable corresponding to the delay (message and processing time) of sending a message from data center a to data center b
- $R \in \mathbb{N}$, stochastic variable corresponding to the number of records inside a transaction
- $X(t) \subseteq \mathbb{N}$, stochastic variable corresponding to the number of expected updates for a given record and a time interval t.
- $W \in \mathbb{R}$, the processing time after reading the value before starting the commit
- $\Theta \in \{commit, abort\}$, stochastic variable corresponding to the commit or abort of a transaction

To simplify the model, we assume that transactions and records inside a transaction are independent. We further assume that all previous transactions are successful. Thus, we might under-estimate the success rate if aborts are more common. The protocol defines that a transaction is committed if all options are successfully learned. We can therefore estimate the likelihood of a commit by calculating the likelihood of successfully learning every option. In turn, learning an option can only be successful if no concurrent option is still pending (i.e., not committed). The first goal is therefore to derive a stochastic variable describing the required time for a previous transaction to commit and become visible.

A transaction for a record r is in conflict only from the moment it requested the option at the leader until it becomes visible, shown as steps (2) to (5). The leader executes the Paxos round by sending a *phase2a* message to all storage nodes and waiting for their *phase2b* responses. We can model the message delay as a stochastic variable $M^{l,b}$ which simply adds the two stochastic variables for sending and receiving the *phase2a* and *phase2b* message from the leader's data center l to some other data center b:

$$M^{l,b} = M^{l \to b}_{phase2a} + M^{b \to l}_{phase2b} \tag{1}$$

The combined distribution for the round-trip requires convoluting the distributions for *phase2a* and *phase2b*. However, the leader only needs to wait for a majority q out of N responses. Assuming, that the leader sends the learning request to all N data centers, waiting for a quorum of answers corresponds to waiting for the maximum delay for all possible combination for picking n out of N responses and can be expressed as:

$$Q^l = \left\{ max\left(x_1 M^{l,1}, \dots, x_N M^{l,N}\right) \mid x_i \in \{0,1\}; \sum_{i=1}^{N} x_i = q \right\} \tag{2}$$

Deriving the distribution for Q^l requires integrating over the maximum of all possible combinations of $M^{l,b}$. After the option has been learned successfully, we can model the message delay to notify the transaction manager c_p by adding the stochastic variable $M^{l \to c_p}_{learned}$ describing the delay to send a learned message:

$$Q^{l,c_p} = Q^l + M^{l \to c_p}_{learned} \tag{3}$$

Unfortunately, even though we now reflect the time of learning an option for a single record, the transaction is only committed if all the options of the transaction are successfully acquired. This entails waiting for the learned message from all involved leaders $(l_1, \dots, l_r)$, with r being the number of updates. Furthermore, after the transaction manager received all learned messages, the update only becomes visible, if the *commit visibility* message arrives at least at the data center of the current transaction c_c before a local read is done for the record (here, we always assume local reads). Given the locations of the leaders $(l_1, \dots, l_r)$ and the previous client's data center c_p, we can model the delay as taking the maximum of all Q^{l,c_p}, one for each leader, and adding the commit message time:[4]

$$U^{c_c}\left(c_p, (l_1, \dots, l_r)\right) = max\left(Q^{l_1,c_p}, \dots, Q^{l_r,c_p}\right) + M^{c_p \to c_c}_{commit} \tag{4}$$

Up to now, we modeled the stochastic variable describing the time to make an update visible for any given record. However, once the update is visible the current transaction still needs to read it and send its option to the leader. Therefore, we first add the transaction processing time w and afterwards add the stochastic variable for sending the *propose* message to the leader l.

$$\Phi^{c_c,l}(c_p, (l_1, \dots, l_r)) = U^{c_c}(c_p, (l_1, \dots, l_r)) + w + M^{c_c,l}_{propose} \tag{5}$$

Note that w is not a stochastic variable. Instead w is the measured time from requesting the read over receiving the response until committing the transaction.[5]. This allows us to factor W out of equation 5 and only consider it in the next step.

[2] Actually, if messages come from the same storage nodes, they can be batched together. However, we do not consider this optimization further as it mainly improves bandwidth.

[3] We describe a stochastic variable in the short form $X \in \mathbb{R}$ instead of defining the function $X : \Omega \to \mathbb{R}$

[4] We changed the subscripts to a function input to express the dependency of c_p

[5] Actually the read-request is not part of the crucial path and should be included. For simplicity, we consider it as part of W as it is a local message delay without any big impact

Given the location c_p of the transaction manager of the previous transaction and all the involved leaders $(l_1, \ldots, l_r)$, $\Phi^{c_c,l}$ describes the time in which no other update should arrive to allow the current transaction to succeed. Unfortunately, these values are unknown. We therefore need to iterate over all possible instantiations of c_p and $(l_1, \ldots, l_r)$ and consider their likelihoods. By assuming independence between the inputs, we can describe the likelihood to finish in time t as:

$$P^{c_c,l}(t) = \sum_{\substack{\tau \in \mathbb{N} \\ l_1 \ldots l_\tau, c_p \in 1 \ldots N}} \Big\{ P(R=\tau) P(L_1 = l_1) \cdots P(L_\tau = l_\tau) \\ P(C_p = c_p) P\Big(\Phi^{c_c,m}\big(c_p, (m_1, \ldots, m_\tau)\big) = t\Big) \Big\} \tag{6}$$

For a single record transaction, the likelihood of committing the transaction is equal to the likelihood of successfully learning the option. Given the likelihood $P(X(t) = 0)$ of having zero other updates in the time interval t, we can express the likelihood of committing the current transaction by multiplying the likelihood of finishing in time t and having no update in t for all possible t:

$$P^{c_c,l}(\Theta = commit) = \int_0^\infty P(X(\gamma) = 0) P^{c_c,m}(\gamma)\, \mathrm{d}\gamma \tag{7}$$

Since w is a constant, we can make all stochastic variables up to this point independent of the current transaction, by factoring out w from Φ and considering it as part of the time as:

$$\Phi_W^{c_c,l}(c_p, (l_1, \ldots, l_r)) = U^{c_c}(c_p, (l_1, \ldots, l_r)) + M_{propose}^{c_c,l} \tag{8a}$$

$$P^{c_c,l}(\Theta = commit) = \int_w^\infty P(X(\gamma) = 0) P^{c_c,m}(\gamma - w)\, \mathrm{d}\gamma \tag{8b}$$

Finally, in order to generalize the likelihood of committing a single record transaction to a transaction with multiple records, we need to calculate the likelihood that all updates are acquired successfully. Assuming independence between records, this can simply be done by multiplying the likelihood of success for each record φ inside the transaction:

$$P^{c_c,(l_1 \ldots l_\varphi)}(\Gamma = commit) = \prod^{\varphi} P^{(c_c,l)}(\Theta = commit) \tag{9}$$

Note that $P^{c_c,(l_1 \ldots l_\varphi)}$ assumes that we know the current data center c_c as well as all involved leaders $(l_1 \ldots l_\varphi)$. However, in contrast to the previous transaction, we have all this information for the current transaction as we know the involved records.

Even though it may look complicated to derive all the distributions for the various message delays and to do the actual computation, it turned out to be straightforward. This is mainly due to the fact that most of the measured statistics and convolution computations are independent of the current transaction and can be done off-line instead of online. Furthermore, it turned out, that we can simplify some of the distributions as the variance does not play an important role. We describe the implementation of the model and the simplifications in Section 5.2.

5.1.3 Other Protocol Models

Even though we only showed the model for the protocol of [24], it should be obvious that it is possible to model the conflict rate for other systems as well. For example, we could use a similar model as proposed in [7] to model the likelihood of losing updates in a typical eventually consistent, quorum protocol as used by distributed key/value stores, Cassandra or Dynamo [15]. Furthermore, we could restrict the model to Megastore by assuming updates per partition instead of per record. Finally, the model could be adapted slightly to model more classical two-phase commit implementations by introducing extra wait delays.

5.2 System Statistics and Computations

We only had to make small changes to an existing implementation of MDCC, in order to support the PLANET language. Most of the changes to the system collect statistics on the characteristics of transactions. By gathering statistics on various attributes of the deployed system, the measurements can be used to calculate useful estimations such as estimated duration or commit likelihood, using the model in Section 5.1.

The model in Section 5.1 defines various required statistics. However, most of the statistics can be collected on a system-wide level and be approximated. Specifically, the distribution of the stochastic variable of equation 8a can be entirely pre-computed for all possible master/client configurations. Afterwards, given the current number of records inside the current transaction, their leader location, and the update arrival rate per record, the computation of final probability reduces to a look-up to find the distribution of the stochastic variable from data center c_c to master l and integrating over the time as shown in equations 8b and 9. Furthermore, in practice, the integration itself is simplified as we use histograms for the statistics. In this section, we describe the required statistics.

5.2.1 Message Latencies

Instead of individual per message type statistics, we simplified the model and assume that the message delays are similar for all message types. Hence we only measure the round trip latencies between data centers by sending a simple RPC message to storage nodes in all the data centers. The clients keep track of histograms of latencies for every data center. In order to disseminate this information to other clients in the system, the clients send their histograms in the RPC message to the storage nodes. The storage nodes aggregate the data from the different clients and data centers and send the information back with the response to the clients. Furthermore, we implemented a window based histogram approach [23], and age out old round trip values, in order to better approximate the current network conditions.

5.2.2 Transaction Sizes

The distribution of transaction sizes is collected in a similar way as the data center round trip latencies described in the previous section. Whenever a transaction starts, the application server stores the size in a local histogram, and occasionally distributes the histograms by sending the data to some storage nodes throughout the distributed database. The distribution of transaction sizes is useful for estimating transaction durations and is used in the conflict estimation models.

5.2.3 Record Access Rates

The likelihood equation 8b needs the likelihood of zero conflicting updates. As a simplification, we assume the update-arrival rate follows a Poisson process, so it is sufficient to compute the average arrival rate as the λ parameter.

The update-arrival rate for individual records is measured on the storage nodes. On the servers, the number of accesses to a particular record is counted in *bucket* granularities, to reduce the size of the data. Also, only the most recent buckets are stored for each record to reduce the storage overhead. In our system, the configured size of a *bucket* is 10 seconds, and only the 6 latest buckets are maintained, which we aggregate using the arithmetic mean. Using these buckets of accesses, the arrival rate described in Section 5.1 can be calculated without significant space overhead ([23] describes in more detail the required overhead).

5.2.4 Computations

Given the message latencies and the transaction size statistics, it is possible to convolute all these statistics according to equation 8a.

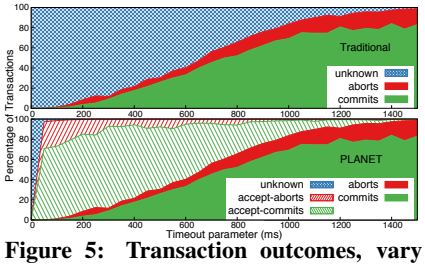

Figure 5: Transaction outcomes, vary timeout (20,000 items, 200 TPS)

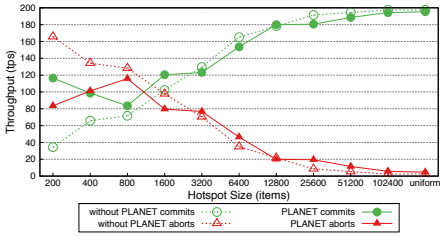

Figure 6: Commit & abort throughput, vary hotspot (200,000 items, 200 TPS)

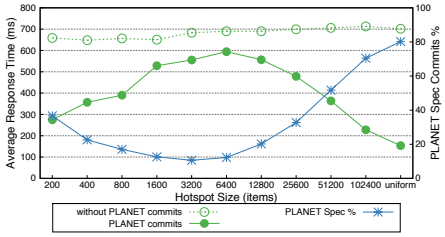

Figure 7: Average response time, vary hotspot (200,000 items, 200 TPS)

The result is an $N \times N$ matrix with one entry per data-center pair. This matrix is very compact and can be stored on every storage node and client, allowing a very efficient likelihood computation.

Whenever a transaction is started, it uses the current statistics and pre-computed values, and computes the commit likelihood of the transaction, using equation 9. The computation is negligible, and adds virtually no overhead compared to the overall latencies between data centers. Also, whenever a new message or response is received, the client re-computes the likelihood with the new information. This allows the commit likelihood to become more accurate as more information is revealed during the commit process.

6. EVALUATION

We evaluated PLANET on top of MDCC, deployed across five different data centers with Amazon EC2. Our evaluation shows that (1) PLANET significantly reduces the uncertainty of transactions with timeouts, (2) speculative commits and admission control notably improve the overall throughput and latency, (3) our prediction model for MDCC is accurate enough for various conflict rates, and (4) the dynamic admission control policy generally provides the best throughput for a variety of configurations.

6.1 Experimental Setup

We implemented PLANET on top of a publicly available implementation of MDCC [24, 20] from UC Berkeley, which we modified according to Section 5.2. We deployed the system in five different data centers of Amazon EC2: US West (Northern California), US East (Virginia), EU (Ireland), Asia Pacific (Tokyo), Asia Pacific (Singapore). Each data center has a full replica of the database (5 times replicated), partitioned across two m1.large servers per data center. Clients issuing transactions are evenly distributed across all five data centers, and are on m1.large servers, separate from the data storage nodes. Clients behave in the open system model, so they issue transactions at a fixed rate, in order to achieve a global target throughput. For all the experimental runs, clients ran for 3 minutes after a 2 minute warmup period, and recorded throughput, response times, and statistics. We further configured PLANET to consider a transaction as accepted as soon as the first storage node confirmed the transaction proposal message.

6.2 TPC-W-like Buy Transactions

We use a TPC-W-like benchmark for all of the experiments. TPC-W is a transactional benchmark which simulates clients interacting with an e-commerce website. TPC-W defines several read and write transactions, but for our purposes, we only test the TPC-W order buying transaction. Many TPC-W transactions focus on reads, which are orthogonal to the transaction programming model. Our buy transaction randomly chose 1–4 items, and purchases them by decrementing the stock levels (similar to the code shown in Listing 2). To focus on the database transaction, we forego any credit card checks, etc., and focus on a single Items table with the same attributes as defined in the TPC-W benchmark.

6.3 Reducing Uncertainty With PLANET

Using PLANET's *onAccept stage block* can reduce the amount of uncertainty that applications may experience. To evaluate the effectiveness, we used the buy transaction with varying timeouts from 0ms to 1500ms. The clients used a uniformly random access pattern, and the fixed client rate was set to 200 TPS, for moderate contention. The Items table had 20,000 items, and both speculative commits and admission control were disabled.

Figure 5 shows the breakdown of transaction outcomes for the different timeout values, for PLANET and a traditional JDBC model with normal timeouts.[6] The solid areas of the graph show the percentage of transactions of which the outcomes are known when the timeout expires. All other portions of the graph (striped, cross-hatched) represent transactions which have not finished before the timeout. For the traditional model, there can be a large percentage of transactions with an unknown state when the timeout expires (blue crosshatched area in top graph). So, for a given timeout value, a larger crosshatched area means the application is more frequently in the dark and more users will be presented with an error.

With PLANET, transactions quickly reach the accepted stage, with the promise that they will eventually complete and the user will be informed of the final outcome. The *accept-commits* and *accept-aborts* areas of the PLANET graph are those transactions which were *accepted* before the timeout, but completed after the timeout. By being notified through *finallyCallback*, applications can discover the true outcome of transactions even if they do not complete before the timeout, and this can drastically reduce the level of uncertainty. For the red and green striped areas, the application can present the user with a meaningful message that the request was received and that the user will be notified about the final outcome, instead of showing an error message. Furthermore, in contrast to the traditional model, for the transactions which only reach the *onFailure* stage within the timeout (blue crosshatched area in the bottom graph), the finally callbacks may be invoked as long as the transaction is not actually lost due to a failure.

As Figure 5 shows, PLANET is less sensitive to the timeout parameter using *onAccept* and *finallyCallback*, because it allows for providing more meaningful responses to the user and for learning the transaction outcome even after the timeout.

6.4 Overall Performance

In order to show the overall benefits of PLANET, we used the TPC-W-like buy transaction, with the clients executing at a fixed rate, to achieve a fixed target aggregate throughput. The transactions used a 5 second timeout, and no *onAccept* stage. To simulate non-uniform access of very popular items, we used a hotspot access pattern, where 90% of transactions accessed an item in the hotspot.

[6]The transaction latencies are not a property of PLANET itself but of the underlying database system MDCC. Also, all presented latencies in this paper are not directly comparable to the latencies shown in [24] as [24] focuses on MDCC optimizations such as fast Paxos and commutativity, instead of the classic protocol, and uses different workloads (no contention, less load, etc).

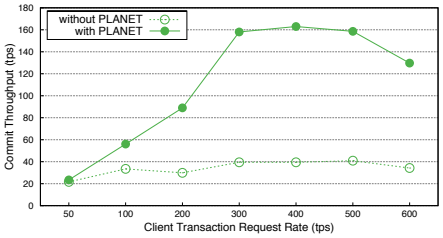

Figure 8: Commit throughput, vary client rate (50,000 items, 100 hotspot)

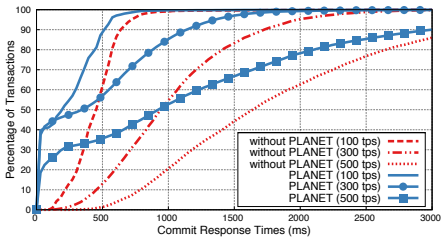

Figure 9: Commit response time CDF (50,000 items, 100 hotspot)

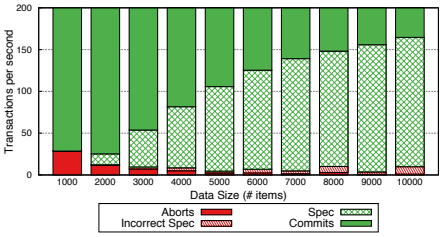

Figure 10: Transaction types, vary data size (200 TPS, uniform access)

The Items table had 200,000 items and we varied the hotspot size to vary the transaction conflict rates. The PLANET system enabled admission control with the *Dynamic(50)* policy, so when the likelihood of commit, L, is less than 50%, then the transaction is attempted with probability L. The PLANET transaction also enabled speculative commits with a value of 0.95, so when the likelihood of commit is at least 95%, the transaction is considered committed.

Figure 6 shows the commit and abort rates for different hotspot sizes, with a client target throughput of 200 TPS. The figure shows that as the hotspot size grows (decreasing conflict rates), PLANET achieves similar throughput with the standard system, with abort rates around 1%–2% with the uniform access. As the hotspot sizes shrink (i.e., more conflicts are created), the abort rates increase because of the increasing conflict rates, and PLANET begins to experience higher commit throughput. When the hotspot is 200 items, PLANET has a commit rate of 58.2%, but the standard system only has a commit rate of 17.1%. The better commit rate is explained by the *Dynamic(50)* policy, which has the biggest impact at roughly 800 or fewer hotspot items. We study the effect of different admission control parameters in Section 6.7.

Figure 7 shows the average commit response times for the different target throughputs. The PLANET response times are all faster than those of the standard system, because PLANET can take advantage of faster, speculative commits. As the hotspot size increases from 200 to 6400 items, the average PLANET response times (green solid line) increase because reducing the conflict rate in the hotspot increases the commit likelihoods and fewer transactions will be rejected. This means fewer transactions will be able to run in the less contended portion of the data, and be able to experience speculative commits. However, as the hotspot size increases further from 6400 items, the hotspot is then large enough where even the transactions accessing the hotspot begins to experience faster, speculative commits and the response time decreases again. Figure 7 also shows the percentage of commits which are speculative, and demonstrates that PLANET response times are low when a larger percentage of speculative commits are possible.

Overall, the throughput and response time are significantly improved by PLANET. The next sub-sections study in more detail the impact of contention, speculative commits and admission control on the overall performance.

6.5 Performance Under High Contention

To further investigate the performance under high contention, we varied the client request rates (i.e., Client Transaction Rate) with a fix set of Items (50,000) and hotspot (100 items). Figure 8 shows the successful commit transaction throughput (i.e., goodput) of the PLANET system , for various client requests rates. The PLANET system outperforms the standard system for all the client throughputs and achieves up to 4-times more throughput at higher requests rates. For the standard system, the throughput peaks at around 40 TPS, where as with PLANET, the peak throughput is around 163 TPS. The abort rates for PLANET ranged from 44% to 75.8% at 600 TPS (hence, the difference between request rate and actual commit throughput). Without PLANET, the abort rates ranged from 56.7% to 94.1% at 600 TPS. Again, PLANET admission control is the main reason for the improved goodput. Admission control prevents thrashing the system, uses resources to attempt more likely transactions, improves commit throughput, and improves the goodput within the hotspot by reducing the contention.

Figure 9 shows the cumulative distribution functions (CDF) for committed transaction response times for experimental runs of various client request rates. It shows that the latencies for PLANET transactions are lower than the latencies for transactions not using PLANET. The main reason for the reduced response times with PLANET is that speculative commits are utilized. At 300 TPS, about 46.2% of all commits could commit speculatively, therefore greatly reducing response times. Many of the transactions not in the hotspot (cold-spot) are able to commit speculatively, with a commit likelihood greater than 0.95 because of the low contention. Therefore, the commit likelihoods of cold-spot transactions are high. At low load of 100 TPS, 95.5% of transactions in the cold-spot could commit speculatively, and at high load of 500 TPS, about 20.2% of transactions were speculative commits. These results show that the speculative commits of PLANET can improve the response times.

6.6 Speculative Commits

In order to study our prediction model in more detail, we ran experiments with our benchmark, but with the transaction size at 1 item, a 5 second timeout, and no *onAccept* stage. To better evaluate only the speculative model, we forego admission control and used a more balanced contention scheme by selecting uniformly items from the Items table, which we varied in size from 1,000 to 10,000 items. The transactions were defined to speculatively commit when the likelihood was at least 0.95 and we used a client transaction request rate of 200 TPS.

Figure 10 shows the breakdown of the different commits types, for the different data sizes. In the figure, standard commits are labeled as "Normal", speculative commits are labeled as "Spec", and speculative commits which are incorrect are labeled as "Incorrect Spec". When the data size is large and there is low contention on the records (10,000 items), most of the transactions can commit speculatively. At 10,000 items, about 77.3% of transactions could commit speculatively because of the high likelihood of success. When the data size is small and there is high contention (1,000 items), most of the transactions cannot take advantage of speculative commits. At 1,000 items, only 0.1% of transactions could commit speculatively. This occurs because as contention increases, the records have higher access rates, so it becomes less likely that a transaction would have a commit probability of at least 0.95.

Since speculative commits finish the transaction early, before the final outcome, sometimes the commit can be wrong. It is clear in Figure 10, that the fraction of incorrect commits is not very large. The transaction defines speculative commits with a likelihood of at least 0.95, so ideally only about 5% of speculative commits would be incorrect. For all the data sizes greater than 1,000 items, the rates of incorrect speculative commits were between 1.8% and 5.8%.

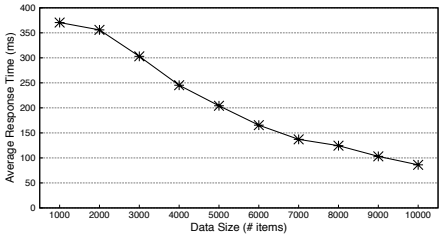

Figure 11: Average commit latency, vary data size (200 TPS, uniform access)

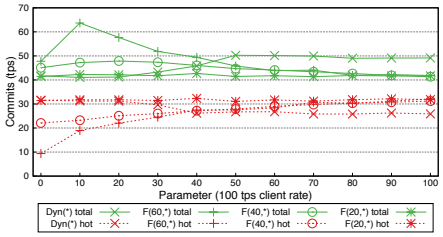

Figure 12: Admission control, vary policies (100 TPS, 25,000 items, 50 hotspot)

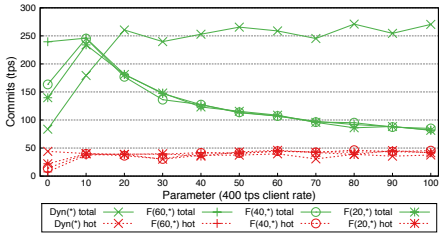

Figure 13: Admission control, vary policies (400 TPS, 25,000 items, 50 hotspot)

For 1,000 items, 25.6% of speculative commits were incorrect. The higher error rate for 1,000 items can be explained by the fact that not many transactions commit speculatively (only 39 in 3 minutes), and high contention makes it difficult to predict the commit likelihood accurately. However, most of the error rates are similar to, or better than the expected rate of 5%. We also studied the prediction model for transactions with more than one record as well as non-uniform access (different conflict rates). The results were very similar to the ones in Figure 10 and not shown.

Figure 11 shows the average transaction response times (including aborts) for the same setup as Figure 10. The graph shows the expected: larger data sizes lower the response times because more transactions can commit speculatively. We conclude that even our simple prediction model provides enough accuracy and is able to significantly lower the total response times.

6.7 Admission Control

Finally, we studied the effects of PLANET's admission control more closely, by running experiments with our benchmark with smaller data sizes, and without speculative commits. Transaction size was at 1 item, the data size was set to 25,000 items, and the hotspot size was set to 50 items. We ran the experiments with different client request rates, and varied the parameters for the *Fixed* and *Dynamic* policies, to observe how the parameters are affected by different access rates. For *Fixed(threshold,attempt_rate)*, we varied the *attempt_rate*, for a few constant values of the *threshold*. For *Dynamic(threshold)*, we varied the *threshold*.

Figures 12 and 13 show the commit rates for the policies with client throughputs of 100 TPS and 400 TPS, respectively. We tested the *Dynamic(*)*, *Fixed(20,*)*, *Fixed(40,*)*, and *Fixed(60,*)* policies, represented by *Dyn(*)*, *F(20,*)*, *F(40,*)*, and *F(60,*)* in the graphs, where "$*$" refers to the parameter we varied (X-axis). The figures show the total commit rates (solid green lines), and the hotspot commit rates (dashed red lines), while varying parameters.

In general, for a 100 TPS request rate (Figure 12) all policies behave similarly. At 100 TPS, the contention level is not strong enough for the admission control policies to really show an impact. However, they are three configurations, which stand out: *Fixed(60,*)*, *Fixed(40,*)* and *Dyn(*)*. *Fixed(60,*)* and *Fixed(40,*)* overcompensates for *attempt_rate* near 0% as they reject too many hotspot transactions causing them to drop significantly below the maximum hotspot throughput of 30 TPS. However, with an attempt rate of 10%, *Fixed(60,10)* achieves the highest total throughput in this setup. The reason is, that *Fixed(60,10)* is too aggressive in rejecting low commit likelihood transactions and influences the workload to a more uniform workload, whereas the other policies still attempt and commit more transactions in the hotspot region (around 30 TPS). This is undesirable, since the admission control technique should fully utilize the hotspot instead of underutilizing it. In contrast, the *Dyn(*)* always utilizes the hotspot at around 30 TPS while providing a good overall throughput.

For a 400 TPS request rate (Figure 13) the situation is different. The dynamic policy performs poorly with a threshold near 0%, whereas the *Fixed* strategies do well, the high threshold (60%) in particular. The reason is simple. Recall a *Fixed(T,A)* policy means that when the commit likelihood is less than T%, the transaction will be attempted A% of the time. With a setting of *Fixed(60,0)* the admission control is most aggressive, whereas the *Fixed(60,10)* will admit some transactions accessing the hotspot and increase the overall performance. In contrast, a *Dyn(0)* policy actually refers to a setup without any admission control. It is more appropriate to compare the *Dyn(60)* point with all the data points of *Fixed(60,*)*, as a *Dyn(60)* policy means that all transactions with a likelihood L, less than 60% will be attempted L% of the time.

In general, the *Dynamic(100)* policy, which means that all transactions are tried in proportion to their commit likelihood, performed very well in both experiments. This also shows that our prediction model is accurate enough to allow *Dynamic* to make good decisions. Lower thresholds for the *Dynamic* strategy essentially accept more risky transactions into the system. In our setup, the *Dynamic* policy performed similarly for all thresholds greater than 50%.

For all configurations, using admission control always resulted in a higher total commit rate than when not using admission control. With admission control, PLANET can back off from the hotspot, not spend resources and thrash the commit protocol, and try to execute transactions which have a much better chance to succeed (transactions not accessing the hotspot). These experiments show the sensitivity analysis for the parameters and policies, and show that the default policy for PLANET, *Dynamic* with a high threshold, works well for a range of environments.

In summary, PLANET enables application developers to write responsive applications by speculatively committing transactions and using the *onAccept* stage. Also, when the system performs admission control with commit likelihoods, resources can be spent for transactions more likely to commit, resulting in higher throughput.

7. RELATED WORK

The PLANET transaction programming model enables application developers to write latency sensitive applications in high variance environments. In [19], the authors describe three types of design patterns in eventually consistent and asynchronous environments: memories, guesses, and apologies. PLANET supports these design patterns with speculative commits, and finally callbacks.

This paper described a conflict estimation model in to predict the likelihood of commit for a write-set, optimistic concurrency control system. Similarly, the work in [34] developed models for two-phase locking, and the models can be implemented in PLANET to compute the commit likelihoods, for PLANET to work in systems using two-phase locking.

[29, 32, 23] all use probabilistic models to limit divergence of replicas or inconsistencies with respect to caching. PLANET uses probabilistic models to predict the likelihood of commit, instead of possible data inconsistencies or divergence.

There have been many studies on transaction admission control, or load control in databases. In [18], the "optimal" load factor is approximated by adaptively probing the performance of the system

with more or less load, and admission control prevented thrashing of the system. [10, 28, 33] have all studied the effects of thrashing and admission control with two-phase locking concurrency control. These solutions require keeping track of the global number of transactions or locks held by transactions to make admission control decisions, which is difficult in geo-replicated distributed databases. The authors of [17] implemented a proxy for admission control by rejecting new transactions which may surpass the system capacity computed offline. PLANET differs from these solutions by using commit likelihoods to make decisions on admission control.

There have been previous proposals for system implementations that perform optimistic commit, sometimes needing compensation if the optimistic decision was wrong [26, 21]. PLANET is orthogonal to this, with speculative commits at the language level that allow the application programmer awareness of the commit likelihood, so a principled decision can trade-off the benefits of fast responses against the occasional compensation costs. Any optimistic commit protocol could be used to implement the PLANET model, so long as PLANET can predict the probability of eventual success.

Several systems support time-outs for transactions, such as JDBC drivers or Hibernate, but to our knowledge, they only support simple timeouts with no further guarantees. Also, with most models, after the timeout expires, the transaction outcome is unknown without an easy way to discover it. Some models allow setting various timeouts for different stages of the transaction (e.g., Galera, Oracle RAC), but how the timeouts effect the user application is not obvious. In contrast, PLANET provides a solid foundation for developers to implement highly responsive transactions using the guess and apology pattern, as well as minimizes the uncertain state for which the application does not know anything about the transaction.

8. CONCLUSION

High variance and high latency environments can be common with recent trends towards consolidation, cloud computing, and geo-replication. Transactions in such environments can experience unpredictable response times or unexpected failures, and the increased uncertainty makes developing interactive applications difficult. We proposed a new transaction programming model, Predictive Latency-Aware NEtworked Transactions (PLANET), that offers features to help developers build applications. PLANET exposes the progress of the transactions to the application, so that it can flexibly react to unexpected situations while still providing a predictable and responsive user experience. PLANET's novel commit likelihood model and user-defined commits enable developers to explicitly trade-off between latency and consistent application behavior (e.g., apologizing for moving ahead too early), making PLANET the first implementation of the previously proposed *guesses* and *apologies* transaction design pattern [19]. Furthermore, likelihoods can be used for admission control to reject transactions which have a low likelihood to succeed, in order to better utilize resources and avoid thrashing. We evaluated PLANET in a strongly consistent, synchronous geo-replicated system and showed that using the features of PLANET can improve the throughput of the system, and decrease the response times of transactions.

When implemented in a strongly consistent, synchronous geo-replicated system, PLANET can offer the lower latency benefits of eventual consistency. While eventually consistent systems choose to give up data consistency or multi-record transactions for improved response times, PLANET can improve transaction response times while still keeping data consistent.

Acknowledgements: We would like to thank the SIGMOD reviewers for their helpful feedback. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Apple, Inc., Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, GameOnTalis, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

9. REFERENCES

[1] Amazon EC2 Outage, April 2011. `http://aws.amazon.com/message/65648/`.

[2] Amazon RDS Multi-AZ Deployments. `http://aws.amazon.com/rds/mysql/#Multi-AZ`.

[3] Google AppEngine High Replication Datastore. `http://googleappengine.blogspot.com/2011/01/announcing-high-replication-datastore.html`.

[4] Hibernate. `http://www.hibernate.org/`.

[5] PLANET. `http://planet.cs.berkeley.edu`.

[6] M. Armbrust et al. PIQL: Success-Tolerant Query Processing in the Cloud. *PVLDB*, 5(3):181–192, 2011.

[7] P. Bailis et al. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow.*, 5(8), 2012.

[8] J. Baker et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[9] P. Bodik et al. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proc. of SoCC*, 2010.

[10] M. J. Carey, S. Krishnamurthi, and M. Livny. Load Control for Locking: The 'Half-and-Half' Approach. In *PODS*, 1990.

[11] B. F. Cooper et al. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1, 2008.

[12] B. F. Cooper et al. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of SoCC*, 2010.

[13] J. C. Corbett et al. Spanner: Google's Globally-Distributed Database. In *Proc. of OSDI*, 2012.

[14] C. Curino et al. Workload-Aware Database Monitoring and Consolidation. In *Proc. of SIGMOD*, SIGMOD '11, New York, NY, USA, 2011. ACM.

[15] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of SOSP*, 2007.

[16] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance Prediction for Concurrent Database Workloads. In *Proc. of SIGMOD*, 2011.

[17] S. Elnikety et al. A method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *WWW*, 2004.

[18] H.-U. Heiss and R. Wagner. Adaptive Load Control in Transaction Processing Systems. In *VLDB*, 1991.

[19] P. Helland and D. Campbell. Building on Quicksand. In *CIDR*, 2009.

[20] H. Jayathilaka. MDCC - Strong Consistency with Performance. `http://techfeast-hiranya.blogspot.com/2013/04/mdcc-strong-consistency-with-performance.html`, 2013.

[21] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of ICDCS*, 1999.

[22] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proc. of SIGMOD*, 2010.

[23] T. Kraska et al. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB*, 2(1), 2009.

[24] T. Kraska et al. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[25] L. Lamport. Paxos Made Simple. *SIGACT News*, 32(4), 2001.

[26] E. Levy, H. F. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. In *Proc. of SIGMOD*, 1991.

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proc. of SOSP*, 2011.

[28] A. Mönkeberg and G. Weikum. Conflict-driven Load Control for the Avoidance of Data-Contention Thrashing. In *ICDE*, 1991.

[29] C. Olston, B. T. Loo, and J. Widom. Adaptive Precision Setting for Cached Approximate Values. In *SIGMOD Conference*, pages 355–366, 2001.

[30] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, 3(1-2), 2010.

[31] E. Schurman and J. Brutlag. Performance Related Changes and their User Impact. Presented at Velocity Web Performance and Operations Conference, 2009.

[32] S. Shah, K. Ramamritham, and P. J. Shenoy. Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers. *IEEE Trans. Knowl. Data Eng.*, 16(7):799–812, 2004.

[33] A. Thomasian. Thrashing in Two-Phase Locking Revisited. In *ICDE*, 1992.

[34] A. Thomasian. Two-Phase Locking Performance and Its Thrashing Behavior. *TODS*, 18(4), 1993.