GUESSTIMATE: A Programming Model for Collaborative Distributed Systems

Kaushik Rajan
Microsoft Research India
krajan@microsoft.com

Sriram Rajamani
Microsoft Research India
sriram@microsoft.com

Shashank Yaduvanshi
Indian Institute of Technology, Delhi
hyaduvanshi@gmail.com

Abstract

We present a new programming model GUESSTIMATE for developing collaborative distributed systems. The model allows atomic, isolated operations that transform a system from consistent state to consistent state, and provides a shared transactional store for a collection of such operations executed by various machines in a distributed system. In addition to "commited state" which is identical in all machines in the distributed system, GUESSTIMATE allows each machine to have a replicated local copy of the state (called "guesstimated state") so that operations on shared state can be executed locally without any blocking, while also guaranteeing that eventually all machines agree on the sequences of operations executed. Thus, each operation is executed multiple times, once at the time of issue when it updates the guesstimated state of the issuing machine, once when the operation is committed (atomically) to the committed state of all machines, and several times in between as the guesstimated state converges toward the committed state. While we expect the results of these executions of the operation to be identical most of the time in the class of applications we study, it is possible for an operation to succeed the first time when it is executed on the guesstimated state, and fail when it is committed. GUESSTIMATE provides facilities that allow the programmer to deal with this potential discrepancy. This paper presents our programming model, its operational semantics, its realization as an API in C#, and our experience building collaborative distributed applications with this model.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Design, Languages

Keywords Distributed systems, collaborative applications, concurrency, language extensions

1. Introduction

Programming distributed systems is difficult for several reasons. Latency between different components is high, and thus any communication between components leads to large delays. Concurrency between components is intrinsic, and concurrent programming is inherently hard. The programmer needs to deal with failures of various forms such as nodes, links, and disks. Consistency between components in various machines needs to be traded off with performance. In this paper we propose a programming model GUESSTIMATE, which explores a new design point in the trade-off between consistency and performance.

One way to build a distributed system is to centralize shared data in the server, but this can affect responsiveness due to latency and contention at the server. If we start caching or replicating the shared data locally at each machine, we open a different can of worms —it is non-trivial to keep all copies consistent. Thus, there is an inherent tradeoff between responsiveness and consistency, and there is a lot of prior work in this area. On the one extreme, we have *one copy serializability* [4]. In this model, every action on shared data is committed and made visible to all machines at the same time. One copy serializability is the best form of consistency we can hope for. However, this programming model is inherently slow. The other extreme is *replicated execution*, where each machine has its own local copy of the shared data, and can update the local copies independently, which is very high performance, but there is no consistency between the states of the various machines. Decades of research has been done to explore the tradeoff between these two models including sequential consistency [11], serializability [4], linearizability [9], and eventual consistency [14].

GUESSTIMATE explores a new design point in the consistency-performance tradeoff. In order to allow performance, GUESSTIMATE allows each machine to maintain a local copy of the shared state (called "guesstimated state"). At any point in time, the guesstimated copy in each machine contains the expected outcome of the sequence of operations issued by that machine, assuming no conflicts with operations executed by other machines. Eventually, the sequences of operations on all machines converge to the same value. However, until this happens, each machine proceeds independently with the current guesstimate it has. The advantage of the programming model is that operations can be executed by any machine on its guesstimated state without waiting for any communication with other machines. The disadvantage is that each operation executes multiple times —from the time of issue up to the time of commit the operation updates the guesstimated copy, and finally at the time of commit the operation atomically updates the committed copy of all the machines — and the result of these executions is not guaranteed to be identical, and the programmer has to deal with this discrepancy.

In our model, operations transform shared objects from consistent state to consistent state. Consistency is specified by preconditions on operations. If at the time of issue an operation fails preconditions on the guesstimated state, it is not even issued, and re-

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06. . . $10.00

jected.[1] Otherwise, if an operation has no precondition violations, it is immediately executed on the guesstimated state, without any blocking. The same operation is executed again at commit time on a different copy of the shared state (referred to as committed state) that is guaranteed to be identical in all machines. In between issue and commit of an operation issued by a machine m_i, operations from other machines can commit, and GUESSTIMATE attempts to update the guesstimated state of m_i with the knowledge of these committed operations from other machines. As a results, each operation can be executed several times on the guesstimated state, as it progressively converges toward the committed state. For the class of applications we study, we expect that most of the time the result of the various executions of an operation from issue to commit are identical, but this is not guaranteed.

We believe that the GUESSTIMATE programming model is particularly useful in building collaborative distributed applications, where a group of users are collaborating to perform a shared task, such as solving a puzzle, for example. We have built six such applications with GUESSTIMATE—a multi-player Sudoku game, an event planning application, a message board application, a car pool system, an auction system, and a small twitter-like application (see Section 6). GUESSTIMATE provides facilities by which the programmer can annotate operations with particular kinds of specifications, and use off-the-shelf program verifiers to check that the implementations of the operations satisfy these specifications. These facilities enable the programming model to guarantee that even if there are certain kinds of discrepancies between the results of execution of the operation on the guesstimated state and the actual committed state, as long as these differences do not matter with respect to the specification of the operation, the model can simply tolerate them. For other, more extreme kind of discrepancies which violate the specification of the operation, the model provides completion functions which can be used to inform the user of discrepancies, and have the users deal with such discrepancies. Challenges also arise due to interaction between operations. A user might perform two operations o_1 and o_2 where the inputs to o_2 depend on the results of o_1. Thus, a discrepancy in the execution of o_1 affects the execution of o_2. GUESSTIMATE provides facilities by which such operations with dependencies can be grouped into *atomic* operations with an all-or-nothing execution semantics. In the extreme, the programmer might want one copy serializability for some important operations, and this can be implemented by the programmer explicitly doing blocking until the operations commit.

This paper explores such a programming model, presents an operational semantics for the programming model, a language extension to C# to use this programming model, and our experience in writing some applications with the programming model.

2. Programming with GUESSTIMATE

We motivate GUESSTIMATE's programming model using the example of a multi-player collaborative Sudoku puzzle. The Sudoku puzzle consists of a 9x9 square, where each square needs to be filled with some integer between 1 and 9. The 9x9 square is divided into nine 3x3 sub-squares uniformly. Each instance of the puzzle comes with some pre-populated numbers. The objective of the game is to fill all the entries in the 9x9 square subject to 3 constraints (1) the same number cannot repeat in a row, (2) the same number cannot repeat in a column, (3) the same number cannot repeat in a 3x3 sub-square. Suppose we wish to program a collaborative Sudoku game, where players are distributed across the Internet. That is, we wish to allow different players to collaboratively work on different portions of the puzzle. Maintaining the state of the machine in a server is inflexible, and can lead to poor response times. Thus, we might wish to replicate the state of the puzzle locally in each of the machines.

Programming such an application with replicated state is nontrivial. In each machine, we need to have three components: (1) *model* that maintains the state of the puzzle, (2) a component that manages the user interface, and (3) a *synchronizer* that performs synchronization between the states of various machines. The synchronizer is responsible for communication between all the replicated instances of the model in various machines. Whenever the model changes in any machine, the synchronizer needs to propagate changes to all other machines. There could be conflicts because two updates by two different machines could violate the constraints of Sudoku mentioned above. There could be races because both the UI and synchronizer could simultaneously update the model in each machine. Programming the application correctly taking all these issues into account is difficult. GUESSTIMATE's goal is to hide much of this complexity from the programmer, and offer a simplified programming model where the programmer only needs to worry about expressing the application logic. All the complex logic related to keeping the model synchronized across various machines is handled by the run-time system.

We describe how GUESSTIMATE's programming model can be used to program this application. Figure 1 shows the class `Sudoku`, which represents the model(state shared by multiple computers). In GUESSTIMATE the programmer declares shared objects by deriving from the abstract base class `GSharedObject`. Other than deriving this abstract base class, which forces providing code for a `Copy` method, the programmer can program this class in whatever way she wants. In this case, the class has a 9x9 array `puzzle` declared in line 2. The method `Check` defined in lines 4–10 returns true if on updating `puzzle[row,col]` with `val` the constraints of Sudoku are satisfied, and false otherwise. The method `Update` defied in lines 12–23 updates `puzzle[r,c]` to value v if it is a legal update, and returns true. If the update is not legal, it returns false. Finally the programmer has to define a `copy` method (line 24) that when passed a source object copies the array `puzzle` from the source to the `this` object. Note that other than deriving from the abstract base class `GSharedObject`, the programmer does not have to do anything different from the way she would write this class for a non-distributed application.

Figure 2 shows the user interface class for this application, which uses the `Sudoku` class from Figure 1. In line 6, a new instance of the `Sudoku` class is created using a call to `Guesstimate.CreateInstance`. Internally, GUESSTIMATE creates an instance of the `Sudoku` class and assigns it a unique identifier. Other machines can now share this `Sudoku` instance by calling the `JoinInstance` method. All machines that have joined this Sudoku instance can now update this object while the run-time takes care of all communication and synchronization.

In lines 15–24 of Figure 2 we see the event handler in the UI that is executed in response to an update of row `r`, column `c` with value `v`. In response to this event, in the case of a non-distributed program, this code would simply call `s.Update(r,c,v)`. With GUESSTIMATE, the programmer has to create an operation using `Guesstimate.CreateOperation` as seen in line 17, and issue the operation using `Guesstimate.IssueOperation` as seen in line 19. When the operation is issued, GUESSTIMATE executes it on the guesstimated copy of the `Sudoku` object. If the execution fails due to violation of any of the constraints of Sudoku (that is `s.Update(r,c,v)` returns false), the operation is dropped. Otherwise, GUESSTIMATE submits it internally to a queue, to be committed later simultaneously in all machines. The code in lines 22–23 calls the redraw function if the operation suc-

[1] In addition to better response times, the user gets feedback on operations that fail on the guesstimated state quickly, so that they can potentially alter the operation and resubmit it.

```
1  class Sudoku : GSharedObject {
2    private int[9,9] puzzle;
3    ...
4    private bool Check(int row, int col, int val)
5    {
6      //returns true if adding val at puzzle[row,col]
7      //does not violate the constraints of Sudoku
8      // and false otherwise
9     ...
10   }
11
12   public bool Update(int r, int c, int v)
13   {
14   //adds value to puzzle[r,c] if all constraints
15   // are satisfied
16      if (r > 9 || r <= 0) return false;
17      if (c > 9 || c <= 0) return false;
18      if (v > 9 || v <= 0) return false;
19      if (!Check(r, c, v))
20          return false;
21      puzzle[r, c] = v;
22      return true;
23   }
24   public void copy(GSharedObject src)...
25 }
```

Figure 1. Sudoku class

```
1  public class UI {
2    enum Color {RED, YELLOW, GREEN, BLUE, COLORLESS};
3    Sudoku s;
4    OnCreate()  {
5       //Here is how we create a shared object.
6       s = (Sudoku) Guesstimate.CreateInstance(typeof(Sudoku));
7    }
8    ...
9    void ReDraw(int r, int c, COLOR color){
10     Guesstimate.BeginRead(s);
11     Board[r,c].Text=s.puzzle[r,c];
12     Guesstimate.EndRead(s);
13     Board[r,c].BackColor=color;
14   }
15   void OnUpdate(int r, int c, int v){
16     // the operation we want to do is:  s.Update(r,c,v)
17     Operation op = Guesstimate.CreateOperation( s, ‘‘Update’’, r,c,v);
18     //Issue the shared operation, together with a completion function
19     bool res = Guesstimate.IssueOperation(op,
20                  (bool b) => {if (b) ReDraw(r, c, Color.GREEN);
21                               else ReDraw(r, c, Color.RED)});
22     if(res)
23       ReDraw(r, c, Color.YELLOW);
24   }
25 }
```

Figure 2. UI class

ceeded on the guesstimated copy. The `Redraw` function (lines 9–14) reads out the updated value from the guesstimate copy and refreshes the UI by changing the text on the square and painting it YELLOW. Enclosing reads within `Guesstimate.BeginRead` and `Guesstimate.EndRead` ensures that they are isolated from concurrent writes via the synchronizer.

At the time of commitment, this operation `s.Update(r,c,v)` may succeed or fail. In particular, the operation can fail if between the time instant when `s.Update(r,c,v)` was issued and the time instant when the operation is committed, operations issued by some other machine got committed, and these other operations violate the precondition for the execution of `s.Update(r,c,v)`.

After commitment, GUESSTIMATE calls a completion function, which is supplied as a delegate in the last argument of `Guesstimate.IssueOperation` (lines 20–21). The completion function takes a boolean value as argument. This boolean is the result of executing the shared operation (in this case `Update`) during commitment. The completion operation is used to change the colors on the UI as follows. If the update operation is successful, the completion operation changes the color of the square at row `r`, column `c` to GREEN and if update fails the color is set to RED. The way colors are manipulated by the UI in this example demonstrates how GUESSTIMATE allows the programmer to deal with changes to the shared state if some operations succeeded initially but failed later due to conflicting operations from other machines.
Thus, programming with GUESSTIMATE involves the following steps:

1. Create shared classes (that is, classes for objects that the programmer wants to be shared across multiple machines) such that they are derived from the abstract base class `GSharedObject`. When creating shared objects use `Guesstimate.CreateInstance` or `Guesstimate.JoinInstance` methods provided by GUESSTIMATE.
2. To invoke an operation on a shared object, first create the operation using `Guesstimate.CreateOperation`, and then issue the operation using `Guesstimate.IssueOperation`.
3. Supply a completion function to `Guesstimate.IssueOperation` to reconcile the client of the shared object in case the operation fails during commit.

GUESSTIMATE API. The above example motivates the various aspects of the GUESSTIMATE programming model. We now give a complete description of the GUESSTIMATE API.

- `GSharedObject CreateInstance(Type type)` takes the type of the shared object as input and creates a unique object of that type and returns it to the client. The new object is internally assigned a unique identifier and is registered with GUESSTIMATE.
- `List <string> AvailableObjects()` returns a list of unique identifiers of all objects.
- `Type GetType(string uniqueID)` returns the type of a shared object, given its unique identifier as input.
- `string GetUniqueID(GSharedObject obj)` returns the uniqueID of a shared object.
- `GSharedObject JoinInstance(string uniqueID)` takes the uniqueID of an available object and returns a reference to that object. In, addition the machine executing this operation is registered for future synchronizations on the object.
- `sharedOp CreateOperation(GSharedObject obj, string methodName, params object[] parameterList)` takes a shared object, the method name and the parameters of the method as input and returns a shared operation of type `sharedOp`.

A programmer can combine multiple shared operations to form hierarchical operations. A shared operation can have a hierarchical structure as defined in the grammar below:

$$\begin{array}{lll} \mathit{SharedOp} & := & \mathit{PrimitiveOp} \mid \mathit{AtomicOp} \mid \mathit{OrElseOp} \\ \mathit{AtomicOp} & := & \textbf{Atomic} \; \{ \; \mathit{SharedOp}^* \; \} \\ \mathit{OrElseOp} & := & \mathit{SharedOp} \; \textbf{OrElse} \; \mathit{SharedOp} \end{array}$$

An *AtomicOp* is treated as an indivisible operation with ”all or nothing” semantics. That is, if any one of the enclosed fail, then none of the operations update the shared state. If all the enclosed operations were to succeed then the shared state is updated by all of them. The implementation of **Atomic** operations is done using copy-on-write at the granularity of objects (see Section 4). The *OrElseOp* is a hierarchical operation that allows the programmer to specify an alternate operation in case of failure. The semantics of op_1 **OrElse** op_2 is that at most one of op_1 or op_2 is allowed to succeed with priority given to op_1. If both op_1 ad op_2 fail, then op_1 **OrElse** op_2 fails. The programmer is allowed to arbitrarily nest these two hierarchical constructors. The following API is used to create and issue shared operations:

- `sharedOp CreateAtomic(List <sharedOp> ops)` takes a list of shared operations as input and returns a hierarchical shared operation.
- `sharedOp CreateOrElse(sharedOp op1, sharedOp op2)` takes two shared operations as input and returns a new hierarchical shared operation.
- `IssueOperation(sharedOp s,completionOp c)` takes as first parameter a `sharedOp` s created using `CreateOperation`, `CreateAtomic` or `CreateOrElse`, and a second parameter of type `completionOp`. The type `completionOp` is a delegate type defined with the signature `delegate void CompletionOp(bool v)`. `IssueOperation` applies s to update the guesstimated state immediately. Further, the operation is added to a pending list for committing on all machines. GUESSTIMATE also remembers the machine m that submitted the operation. At the time of commitment the completion operation c is executed on machine m. More details can be found in Section 3.
- `void BeginRead(GSharedObject object)`, `void EndRead(GSharedObject object)` these methods can be used to directly read the guesstimated state without issuing operations. All reads enclosed within `BeginRead` and `EndRead` are guaranteed to be isolated from concurrent writes to the object through the synchronizer.

Apart from using this API to create shared objects, and issue operations on shared objects, the programmer can choose to implement the application as she pleases.

3. Formal Model

In this section, we describe the GUESSTIMATE programming model formally, and give an operational semantics for the model. The programming model is presented at the level of abstraction a programmer needs to understand in order to write programs. Thus, the internal details of how the GUESSTIMATE runtime performs replication and synchronization are somewhat hidden in this presentation. Section 4 presents the GUESSTIMATE runtime and argues that the runtime indeed faithfully implements the operational semantics presented in this section.

Objects and state. A distributed system is a pair $\langle M, S\rangle$, where M is a tuple of $|M|$ machines $\langle m_1, m_2, \ldots, m_{|M|}\rangle$, and S is a set of shared objects. In the example in Section 2, the `Sudoku` object is a shared object. An application can have several such shared objects. A *shared state* is a valuation to the set of shared objects. Σ is the set of all shared states. In addition, each machine $m_i \in M$ maintains a set of local objects L_i, and this set could be different for different machines. A *local state* is a valuation to the local objects. Γ is the set of all local states. The programming model has four kinds of operations: (1) local operation, (2) shared operation, (3) completion operation, and (4) composite operation, described formally below.

The state of a machine is a 5-tuple $\langle \gamma, C, \sigma_c, P, \sigma_g\rangle$, where $\gamma \in \Gamma$ is the *local state* at the machine, C is a sequence of *completed* shared operations, $\sigma_c \in \Sigma$ is the *committed state* at the machine, P is a sequence of *pending* composite operations at the machine, and $\sigma_g \in \Sigma$ is the *guesstimated state* at the machine.

The committed state σ_c is obtained by executing the sequence of completed operations C from the initial state. The programming model guarantees that the sequence C of completed operations is identical across all machines, and thus C and σ_c are equal for all machines in the distributed system (this is an invariant that can be proved by induction on the operational semantics). The sequence P consists of pending operations that have been submitted at this machine, and guesstimated state σ_g is obtained by executing this sequence of operations P starting from the committed state σ_c. Since the sequence P may be different for different machines, the guesstimated state σ_g could be different for different machines. As the system proceeds executing, operations from P are moved to C, and the guesstimated state σ_g and the committed state σ_c converge to the same value when P becomes empty.

The set of machine states is denoted by Δ. The state of the distributed system is a function Π from machine index to Δ. We use the notation $\langle \gamma(i), C(i), \sigma_c(i), P(i), \sigma_g(i)\rangle$ to denote the state of the i^{th} machine $\Pi(i)$.

Operations. Each machine can modify its local state using local operations that can read the guesstimated state and local state, but can update only local state. A *local operation* therefore has signature $(\Sigma \times \Gamma) \to \Gamma$. Local operations do not change shared state, and they are not applied on other machines. In addition to managing local state, local operations can be used to maintain information regarding submitted operations or to query the shared state before applying an operation to change the shared state. The set of all local operations is $\mathcal{L}$.

A *shared operation* reads and updates the shared state. Shared operations can succeed or fail. We can think of each shared operation as having a precondition $\Sigma' \subseteq \Sigma$ and the operation fails if it is attempted to be executed from a state $\sigma \notin \Sigma'$. For example, if a shared operation tries to buy tickets on an airplane, then it can fail if there are not enough seats available. Thus, in addition to updating the shared state, a shared operation returns a boolean value as well. A shared operation therefore has signature $\Sigma \to (\Sigma \times B)$. The set of all shared operations is $\mathcal{S}$.

We associate a specification $\varphi_s \subseteq \Sigma \times \Sigma$ with every shared operation s. A shared operation s *conforms* to specification φ_s if for any pair of shared states σ_1 and σ_2 (1) if $s(\sigma_1) = \langle \sigma_2, \mathbf{true}\rangle$, then $\langle \sigma_1, \sigma_2\rangle \in \varphi_s$, and (2) if $s(\sigma_1) = \langle \sigma_2, \mathbf{false}\rangle$, then $\sigma_1 = \sigma_2$. That is, a shared operation either returns true and satisfies its specification, or returns false and does not modify the shared state. We use specifications on shared operations to reason about applications written using GUESSTIMATE. More details can be found in Section 5.

A *completion operation* reads the local state, shared state, and a boolean value, and updates the local state. A completion operation therefore has signature $(\Sigma \times \Gamma \times B) \to \Gamma$. The set of all completion operations is $\mathcal{C}$.

A *composite operation* o is a pair $\langle s, c\rangle$ where $s \in \mathcal{S}$ is a *shared operation* and $c \in \mathcal{C}$ is a *completion operation*. In a composite operation, the boolean value produced by the shared operation is passed as an argument to the corresponding completion operation. The set of all composite operations is $\mathcal{O}$. The semantics of a composite operation $\langle s, c\rangle$ can thus be understood as:

$$\begin{array}{ll} \lambda(\sigma, \gamma). & \mathbf{let}\ (\sigma_1, b) = s(\sigma)\ \mathbf{in} \\ & \mathbf{let}\ \gamma_1 = c(\sigma_1, \gamma, b)\ \mathbf{in} \\ & (\sigma_1, \gamma_1) \end{array}$$

Given a composite operation $o = \langle s, c\rangle$ we use the notation $[o]$ or equivalently $[\langle s, c\rangle]$ to denote a function with signature $\Sigma \to \Sigma$ with the following semantics:

$$[\langle s, c\rangle] = \lambda\sigma.\mathbf{let}\ (\sigma_1, b) = s(\sigma)\ \mathbf{in}\ \sigma_1$$

We also extend the notation $[.]$ to map a sequence of composite operations to function with signature $\Sigma \to \Sigma$ with the semantics:

$$[(o_1, o_2, \ldots o_n)] = \lambda\sigma.([o_n]([o_{n-1}] \cdots ([o_2]([o_1](\sigma))) \cdots))$$

$$
\begin{array}{llll}
\mathbf{R}1: & \Pi, \quad o \in \mathcal{L} \text{ issued at machine } i & \Rightarrow & \Pi\{\ \gamma(i) := o(\sigma_g(i), \gamma(i))\ \} \\[1em]
\mathbf{R}2: & \begin{array}{l}\Pi, \quad o = \langle s, c\rangle \in \mathcal{C} \text{ issued at machine } i, \\ \qquad s(\sigma_g(i)) = (\sigma, \mathbf{true})\end{array} & \Rightarrow & \begin{array}{lll}\Pi\{ & P(i) & := \mathbf{Append}(P(i), o); \\ & \sigma_g(i) & := [o](\sigma_g(i))\ \ \}\end{array} \\[1em]
\mathbf{R}3: & \Pi, \quad o = \langle s, c\rangle = \mathbf{First}(P(i)) & \Rightarrow & \begin{array}{lll}\Pi\{ & P(i) & := \mathbf{AllButFirst}(P(i)); \\ & C(i) & := \mathbf{Append}(C(i), s); \\ & (\sigma_c(i), \gamma(i)) & := o(\sigma_c(i), \gamma(i)); \\ & \forall (j \neq i).C(j) & := \mathbf{Append}(C(j), s); \\ & \forall (j \neq i).\sigma_c(j) & := s(\sigma_c(j)); \\ & \forall (j \neq i).\sigma_g(j) & := [P(j)](s(\sigma_c(j)))\ \ \}\end{array}
\end{array}
$$

Figure 3. Operational semantics

Operational semantics. Each machine m_i issues a sequence of local and composite operations $(o_1^i, o_2^i, \ldots)$. Local operations are executed locally on each machine, with no communication to other machines. Each composite operation is first executed locally on the guesstimated state of the issuing machine, and queued in the sequence of pending operations. Atomically, a pending operation is picked from the front of a pending queue of some machine, and executed and committed on all machines.

The operational semantics of GUESSTIMATE is given by the three rules in Figure 3. The rules are guarded commands of the form $\Pi, \varphi \Rightarrow \Pi'$, where Π is the current state, φ is a guard (a predicate that needs evaluates to true on current state and inputs for the rule to be enabled), and Π' is the next state. Π' is obtained by updating some components of Π. We use the notation $\Pi\{c_1 := v_1, c_2 := v_2, \ldots\}$ to denote a new state where component c_1 in Π gets updated to value v_1, component c_2 in Π gets updated to value v_2, etc. Any component not specified as updated retains its old value.

To state the operational semantics formally, we need a few definitions. The function $\mathbf{First}(list)$ returns the oldest entry in $list$. The function $\mathbf{AllButFirst}(list)$ removes $\mathbf{First}(list)$ from the list, and returns all the other elements preserved intact in the same order. The function $\mathbf{Append}(list, o)$ returns a new list with o appended as the last element of $list$.

Rule $\mathbf{R1}$ states that a local operation at machine i merely modifies the local state at machine i. Rule $\mathbf{R2}$ states that a composite operation issued at machine i is appended at the end of the pending queue at machine i, and updates the guesstimated state at machine i. Note that this rule has a guard $s(\sigma_g(i)) = (\sigma, \mathbf{true})$, which requires that the operation s succeed before it is added to the pending queue. If $s(\sigma_g(i)) = (\sigma, \mathbf{false})$, then the operation o is dropped.

Rule $\mathbf{R3}$ describes commitment of a shared operation. A shared operation $o = \langle s, c\rangle$ is picked from the front of the pending queue $P(i)$ of some machine i. The operation is removed from the pending queue $P(i)$ of machine i, and moved to the committed queue $C(i)$ of machine i. The committed state $\sigma_c(i)$ and local state $\gamma(i)$ of machine i are updated by executing the operation o. For every machine j, such that $j \neq i$, the shared operation s is also executed to update the committed state $\sigma_c(j)$. The operation is also appended to the committed sequence $C(j)$ and updates the guesstimated state $\sigma_g(j)$ by executing the pending operations $P(j)$ on the updated committed state. The rule $\mathbf{R}3$ does not have a guard which requires that the operation s be successful. The operation is executed regardless of whether the operation s is successful or not. Also, note that the guesstimated state $\sigma_g(i)$ does not need to be updated, since the concatenation of $C(i)$ followed by $P(i)$ is invariant due to the operation —the first element of $P(i)$ is moved to be the last element of $C(i)$. During commitment, the operation s is executed on all machines simultaneously and atomically, and updates the committed state and guesstimated state in each machine. In reality, this takes several rounds of communication between the machines, but the programming model gives the illusion that the commitment happens atomically on all machines. Section 4 gives details on how the runtime implements atomic commitment. During the commitment process, the completion routine c is run on the machine m_i which issued the operation, and if the operation s fails, then the completion routine c will have the opportunity to take remedial action.

Invariants. GUESSTIMATE guarantees that when the system quiesces and the pending queues of all machines is empty, the guesstimated state and the committed state of all the machines converge to the same value.

Formally, every machine state $\langle \gamma, C, \sigma_c, P, \sigma_g\rangle$ satisfies the invariant $[P](\sigma_c) = \sigma_g$. The entire system (which comprises all machine states) satisfies the invariant that for any pair of machines m_i and m_j, we have that $\sigma_c(i) = \sigma_c(j)$ and $C(i) = C(j)$. These invariants can be proved by induction over the transition rules $\mathbf{R}1$, $\mathbf{R}2$ and $\mathbf{R}3$ allowed by the operational semantics.

4. The GUESSTIMATE runtime

Synchronization. The operational semantics given in Section 3 suffices to use GUESSTIMATE. In this section, we describe our specific implementation of synchronization in GUESSTIMATE runtime. In particular, we describe how we simulate the atomic commitment and how the implementation ensures that each operation is executed at most three times. If the reader is merely interested in using GUESSTIMATE, this description can be skipped. However, if the reader is curious about how the GUESSTIMATE runtime guarantees the semantics given in Section 3, we give some details below.

The synchronizer component of the GUESSTIMATE runtime, maintains all the necessary state needed to coordinate among the machines. At each machine m_i it maintains a list of all shared objects in the system (all calls to `CreateInstance`) and a list of all objects that machine m_i is subscribed to (calls to `JoinInstance` made by m_i). For each object that machine m_i is subscribed to, the synchronizer keeps two copies of the object, one to maintain the committed state $\sigma_c(i)$, and another to maintain the guesstimated state $\sigma_g(i)$. It also maintains an ordered list of pending operations $P(i)$ and a list of committed operations $C(i)$. Every operation that is issued by machine m_i via `IssueOperation` is added to $P(i)$.

To communicate between machines the synchronizer uses PeerChannel, a peer-to-peer (P2P) communication technology that is part of .NET 3.5. PeerChannel allows multiple machines to be combined together to form a mesh. Any member of the mesh can broadcast messages to all other members via a channel associated with the mesh. GUESSTIMATE uses two meshes, one for sending signals and another for passing operations. Both meshes contain all participating machines.

Synchronization among the machines is done in a master-slave mode. One of the machines is designated to be the master and is responsible for periodically initiating the synchronization process. The synchronization happens over 3 stages.

1. `AddUpdatesToMesh`. In the first stage the pending operations from all machines are gathered to construct a consolidated pending list $P_{all}(i)$ at each machine m_i. This happens as follows. Starting with the master each machine m_i, on its turn, flushes out all operations in $P(i)$ as triples of the form $\langle \mathit{machineID}, \mathit{operationnumber}, \mathit{operation} \rangle$ via the `Operations` channel to all the other machines, and then passes the turn to the next machine $i+1$ via a confirmation message on the `Signals` channel. We will refer to the time instant at which machine m_i starts flushing operations as $t_{BeginFlush}(i)$ and the time instant at which it completes flushing operations as $t_{EndFlush}(i)$. In the time interval between $t_{BeginFlush}(i)$ and $t_{EndFlush}(i)$ the implementation does not allow any new operations to be issued. Therefore at the time instant $t_{EndFlush}(i)$, $P(i)$ is empty.

   The number of operations sent on the `Operations` channel is also sent along with the confirmation message. As all machines see these confirmation messages and know the number of participating machines, they know the number of operations to expect in $P_{all}(i)$. Once confirmation messages from all the participants are received the master signals the beginning of the second stage, `ApplyUpdatesFromMesh`.

2. `ApplyUpdatesFromMesh`. Each machine m_i waits until it receives all the expected operations and then applies operations from $P_{all}(i)$ to the committed state $\sigma_c(i)$ in lexicographic order of the pair $\langle \mathit{machineID}, \mathit{operationnumber} \rangle$. Once an operation has been applied it is moved to the *Completed* list $C(i)$.

   An operation o in $P_{all}(i)$ can update the state of machine m_i in one of two ways. If the operation $o = \langle s, c \rangle$ were submitted by machine m_i then o updates $(\sigma_c(i), \gamma(i))$ to the value of $o(\sigma_c(i), \gamma(i))$. If the operation were submitted by some machine $j \neq i$ then o updates $\sigma_c(i)$ to $s(\sigma_c(i))$. Once machine m_i has applied all operations in $P_{all}(i)$ it sends an acknowledgment on the `Signals` channel. After sending the acknowledgment the guesstimated state on m_i, namely $\sigma_g(i)$ is updated to $[P(i)](\sigma_g(i))$ to reflect the changes made since the synchronization began. This is done by first copying the committed state to the guesstimated state by calling `Copy` and then applying each operation in $[P(i)]$. We will refer to the time at which machine m_i begins copying the committed state on to the guesstimated state as $t_{BeginUpdate}(i)$ and the time at which the last operation from $P(i)$ has finished updating $\sigma_g(i)$ as $t_{EndUpdate}(i)$. In the time interval between $t_{BeginUpdate}(i)$ and $t_{EndUpdate}(i)$ the implementation does not allow any new operations to be issued.

3. `FlagCompletion`. Once the master receives acknowledgments from all the machines the synchronization is complete. The master can start another synchronization any time after this.

Concurrency control. Within the guesstimate runtime concurrent updates to shared state are possible. Fine grained locks are used internally by the `Synchronizer` to avoid races. These locks are used to ensure the following: (i) operation are queued atomically into the pending list (ii) in the time interval $[t_{BeginFlush}(i), t_{EndFlush}(i)]$ no operations are issued (iii) the execution of an operation on the guesstimated state happens atomically (iv) In the time interval $[t_{BeginUpdate}(i), t_{EndUpdate}(i)]$ no operations are issued. (v) All reads to shared state enclosed within `BeginRead` and an `EndRead` are guaranteed to be atomic. Note that all these blocking synchronization operations are used from within the `Synchronizer` and none of them involve more than one machine. While the current implementation uses pessimistic concurrency control, lock free implementations and optimistic concurrency control could also have been used. For example, the pending list could be implemented as a lock-free queue and updates to the shared state could be serialized with optimistic concurrency control. This would be useful especially if the shared state were large, as it would allow concurrent independent operations to be applied without blocking.

All or nothing semantics are provided for atomic operations using copy on write. The first time an object is updated within an atomic operation a temporary copy of its state is made and from then on all updates within the atomic operation are made to this copy. If the atomic operation succeeds, the temporary state is copied back to the shared state.

Conformance to the operational semantics. It can be shown that there is a simulation relation between the state transitions in the GUESSTIMATE runtime and the rules for operational semantics in Figure 3. The proof for local operations (corresponding to $\mathbf{R}1$) and issuing composite operations (corresponding to $\mathbf{R}2$) are straightforward. For commitment of operations (corresponding to $\mathbf{R}3$), the crux of the argument is that even though commitment takes several rounds of messages, all composite operations submitted in this duration can be thought of as being submitted either before or after the entire commitment operation completes. In particular, all composite operations issued at machine m_i before $t_{BeginUpdate}(i)$ can be thought of as issued before the atomic commit, and all composite operations issued at machine m_i after $t_{EndUpdate}(i)$ can be thought of as issued after the atomic commit. Note that no operations can be issued in the interval $[t_{BeginUpdate}(i), t_{EndUpdate}(i)]$.

Bounded re-executions. A salient feature of the implementation is that though the operational semantics allows an operation to be executed multiple (possibly unbounded) number of times, our implementation of the GUESSTIMATE runtime ensures that an operation is executed at most three times (including issue and commit). We present an argument for why an operation can execute at most three times.

An operation can be submitted at machine m_i either outside the time interval $[t_{BeginFlush}(i), t_{EndUpdate}(i)]$ or within the time interval $[t_{EndFlush}(i),\ t_{BeginUpdate}(i)]$ (note that no operation can be submitted in the intervals $[t_{BeginFlush}(i), t_{EndFlush}(i)]$ and $[t_{BeginUpdate}(i), t_{EndUpdate}(i)]$).

Suppose an operation o is submitted outside the time interval $[t_{BeginFlush}(i), t_{EndUpdate}(i)]$ Then, operation o is executed once during issue. During the next synchronization, the operation o is guaranteed to get committed (because all operations in the pending list are guaranteed to be committed). The operation o is executed once again during commit, thus executing a total of two times.

Suppose an operation o is submitted within the time interval $[t_{EndFlush}(i), t_{BeginUpdate}(i)]$. As before, operation o is executed once during issue. Next, at the time $t_{BeginUpdate}(i)$ operation o is part of the pending list $P(i)$ and all operations that committed during the current synchronization have updated the committed state $\sigma_c(i)$. o is executed for the second time somewhere in the time interval $[t_{BeginUpdate}(i), t_{EndUpdate}(i)]$ to update the guesstimated state and re-establish the invariant $\sigma_g(i) = [p](\sigma_c(i))$. Finally, during the next synchronization, the operation o is guaranteed to commit and executes one more time (as in the previous case), thus executing a total of three times.

Entering and leaving the distributed system. Machines can dynamically enter and leave the distributed system. When an application written with GUESSTIMATE is started up on a new machine the GUESSTIMATE runtime adds the machine to the `Signals` and `Operations` meshes and broadcasts a special message on the signals channel. The master processes this message before the next synchronization and sends the new device both the list of available objects and the list of completed operations. The new device initializes its state based on these and intimates the master, who then begins synchronization. A machine that leaves the system intimates the master and the master removes this machine from the next synchronization onward.

Failures and fault tolerance. When the master starts a new synchronization phase it assumes all machines are active. However, if the synchronization is stalled for more than a threshold duration, the master goes through the log of messages sent on the signals channel to detect the stalling machine. It resends the signal to which the machine failed to respond. If the fault were transient the machine might respond to this resent signal. If the machine still does not respond, the master removes the machine from the current synchronization phase and sends it a restart signal. On receiving the restart signal the machine shuts down the current instance of the application and restarts the application. Upon restart the machine re-enters the system in a consistent state.

Our current implementation is not tolerant to failure of the master. This support can be added by designating a new machine as master if no synchronization messages are received for a threshold duration.

5. Design Patterns

The two main advantages of programming with GUESSTIMATE are simplicity and responsiveness. The programming model is very close to sequential programming in that the programmer creates shared objects, and issues shared operations on them which execute synchronously without blocking on the guesstimated state. The simple (and common) case is when the results of the operation during the commitment phase are the same as the results obtained from the guesstimated state. However, the programmer needs to handle the case when the two results are indeed different. We encountered several such situations in the applications we have written using GUESSTIMATE, and we have been able to handle these using a few design patterns involving specifications, completions, atomic operations and blocking constructs. We describe these design patterns below.

Specifications. As noted in Section 3, we recommend a programming discipline where every shared operation s conforms to a specification $\varphi_s \subseteq \Sigma \times \Sigma$ (recall from Section 3 that s conforming to φ_s means that if s returns **true** then s respects φ_s, and if s returns false then the shared state is unchanged). We have written such specifications φ_s using Spec# [2], and checked that the body of s conforms to φ_s using the Boogie program verifier [3].

Consider a machine m_i that executes a sequence of operations $(s, t, \ldots)$, where each shared operation s conforms to a specification φ_s. Suppose all these operations succeed during execution on the guesstimated state. Suppose also that each of the operations succeeds during commitment (we consider the case where some of these operations fail during commitment later). This means that the committed sequence of operations could be of the form $(o_1^1, \ldots, s, o_i^2, \ldots, t, \ldots)$ where operations o_j^i submitted by other machines are interleaved between the operations $(s, t, \ldots)$. However, due to definition of conformance, the pre and post states of operations s and t in the committed sequence necessarily satisfy φ_s and φ_t respectively.

For example, in our car pool application, a method `GetRide(Event e)` searches through various ride sharing options to get a ride for the user to attend event `e`. The operation succeeds if *some* vehicle has space for the user. However, it may so happen that during the execution of the method on the guesstimated state the user gets a ride on vehicle v_3 and by the time the operation is committed, vehicle v_3 is full. We have written a predicate $\varphi_{\texttt{GetRide}}$ which is satisfied if the user gets a ride on some vehicle, and established that the implementation `GetRide` conforms to $\varphi_{\texttt{GetRide}}$ using Boogie. This ensures that as long as `GetRide` succeeds in the committed sequence, the user will have some ride, though perhaps not in the initial vehicle v_3 that was obtained during execution on the guesstimated state.

This design pattern is easily extended to hierarchical **OrElse** operations. If operations s and t both conform to a specification φ, it can be established that the operation s **OrElse** t also conforms to φ. Thus, the programmer can compose several alternatives to achieve a goal φ using using the **OrElse** constructor, and still respect this design pattern, allowing the flexibility that the operation could succeed using one alternative during the execution on the guesstimated state and another alternative during commitment.

Completions. Writing specifications for each shared operation and checking for conformance greatly simplifies the task of writing completion routines. If we use this discipline, then completion routines can be written using the form:

```
(bool b) =>
      if (b)
        "indicate in UI that the
         operation successfully committed"
    else
        "indicate in UI that the operation failed,
         and ask the user to take remedial action"
```

Note that the completion routine for the `Sudoku` application seen in Figure 2 follows this pattern.

In essence, the above two design patterns split up the responsibility for handling variances between the guesstimated state and actual committed state between the programmer and the user. The programmer codes up several alternatives for achieving the goals of an operation s such that the operation s returns **true** if any of these alternatives can be executed successfully, and ensures that a specification φ_s (derived from the goal of the operation) holds for each of these alternatives using static analysis. If all the alternatives fail during commitment, the completion routine throws up the problem to the user, and asks the user to deal with the failure.

Atomic operations. There are two kinds scenarios where we have found use for atomic operations. The first is obvious —when we want a set of operations to be executed with all-or-nothing semantics. This happens in our event planning application when a user wants to sign up for two events or none.

The second kind of scenario where we have found use for atomic operations is when there is a value dependency between operations. Suppose we have two operations s and t such that a location written by s is read or written by t. Then, there is the possibility that s and t succeed during execution on the guesstimated state, and that s could fail during commitment, and t could succeed. In such situations if the dependence has to be enforced we suggest grouping s and t together as a single atomic operation.

Blocking operations. Finally, there are certain situations where we really want to be sure that an operation commits before executing subsequent operations. We have encountered this situation, for instance, in our event planning application, where we first require a user to login, and we do not want to allow the user to do anything before we are sure that the login has succeeded.

We have been able to program such scenarios by blocking the main thread on issuing the operation and waiting until the completion routine unblocks it. The general template for doing so is as below.

```
res = IssueOperation(loginOperation,
(bool b) =>
```

```
      if (b)
        "release the thread and allow access"
    else
        "release the thread and deny access"
if(res)
"block the thread"
```

In summary, we have presented four design patterns that we have found useful in writing GUESSTIMATE applications. Our experience is that the above design patterns provide simple ways to handle common situations that arise in programming collaborative applications using GUESSTIMATE.

6. Experience

We have built 6 collaborative applications using the GUESSTIMATE programming model. These are a collaborative multi-player Sudoku puzzle, an event planning application, a message board application, a car pool system, an auction system and a small scale twitter application. In all these applications the shared state and shared operations are encapsulated together in a shared object class. This class derives from the abstract base class `GSharedObject` exposed through GUESSTIMATE and implements a `copy` method. For example, the Sudoku class contains a 9x9 array and an update operation that form the shared data and operation respectively. It also implements a copy method that when passed a source object, copies the 9x9 array from the source to the `this` object. At a high level the rest of the application design typically involves having to call appropriate API functions to create new instances of shared objects and issue operations based on user interaction. All applications are written with about 500-700 lines of code. Below we share our experience in developing these applications using GUESSTIMATE by highlighting some interesting design decisions that we made.

Updating local state. While the shared state is kept globally consistent and up-to-date by the runtime, the programmer has to ensure that the local state is kept up-to-date. The state used by the GUI has to not only be updated in response to user actions but also in response to synchronizer events, and this can be done using GUESSTIMATE's completion operations.

Designing completion operations correctly for the multi-player Sudoku was challenging. Our first design of the GUI was as described in Section 2 where initially when an operation is submitted, the color of the square is changed to yellow, and later depending on success or failure the color is changed to green or red respectively. However, this design differentiated successful operations of the current user from the successful operations done by other users, and our users did not like this distinction. Ultimately, we chose to remove the green color all together and depict all successful operations from all users uniformly. Thus, we decided to use special markings only for tentative operations and failing operations. When an operation succeeds at commit time, we use the completion operation to remove tentative markings.

Another interesting design decision was with regards to refreshing the GUI. In most applications the GUI displays only user specific information in detail, while displaying other information only on demand. For example, in event planner the list of activities joined by the user is always on display and is kept up-to-date via completion operations. On the other hand information regarding vacancy status of events is not displayed unless asked for. Therefore it is often sufficient to frequently refresh only the user specific state of the GUI while updating other state less frequently. Completion operations are well suited to do this.

The one exception we found is in Sudoku where it is essential for updates from other users to reflect on the grid as and when they happen. In our initial design, the grid was refreshed every time the user submitted an operation and every time a completion operation was run. However, this alone was not sufficient. The user often did not see updates from operations submitted elsewhere. In our final design we choose to call refresh based on user activity. Every time the user moves the mouse around on a grid or the grid comes to the fore, refresh is called. So as long as the user is active the display is kept up-to-date. Our user experience study suggests that this is an effective solution. Additional API support, that provides a call back for changes to a shared object via remote operations, could provide an alternate solution.

```
1  private void button_signin_Click(object sender, EventArgs e){
2    string usrnm = textBox_username.Text;
3    string passwd = TextBox_password.Text;
4    sharedOp op = Guesstimate.CreateOperation(handle,usrnm,passwd);
5    Semaphore s=new Semaphore(0,1);
6    bool res = Guesstimate.IssueOperation(op,
7              (bool b) =>
8                  {
9                     if (b) { my_name = username; s.Release();}
10                    else  {
11                       register_failed r = new register_failed();
12                       r.ShowDialog(); s.Release();
13                    }
14                 }
15                 );
16   if (!res)
17     this.Close();
18   else{
19       this.Cursor = Cursors.WaitCursor;
20       s.WaitOne();
21       this.Cursor = Cursors.Default;
22       this.Close();
23     }
24 }
```

Figure 4. Sample code to implement a blocking operation

Blocking via completion operations. In five of the applications (all but Sudoku) we needed to implement two functionalities, signin and new user registration, as blocking functions. New user registration is made blocking to ensure that the same username is not simultaneously registered at two machines. And we choose to make signin blocking to ensure that a user is signed in only on one machine at a time.

Blocking is implemented as shown in Figure 4 by waiting on a semaphore until the completion operation releases it. Here, the login operation is created in line 4, and issued in line 6. If the result of the `IssueOperation` is **`true`**, then the issuing thread simply waits on semaphore s at line 20. The wait is released by the completion routine executing `s.Release()` in line 9. Apart from these two functionalities the rest of the operations in all applications are better suited to a non-blocking design.

Atomic and OrElse Operations. Atomic operations and **OrElse** operations are used extensively in the event planner application. Users can choose to join one among many events and we implemented this using an OrElse operation. *Atomic* operations are used when a user wants to perform multiple operations with all-or-nothing semantics, for example a user chooses to go to a party only if she also gets a ride to the party. *Atomic* operations are also used when there is a value dependence via the shared state. In the event planner, a request to join an event can fail either because there is no more vacancy in the event or because the user has already joined the maximum allowed events. In case a user wants to join an important event ($event_a$), but cannot because she is already used her quota, she might want to leave some other event ($event_b$) and join $event_a$. However, she wants to retain $event_b$ unless she can join $event_a$ for sure. We use *Atomic* operations to ensure that such dependencies are respected during execution.

Specifications and contracts. We designed all classes that implement `GSharedObject` in Spec#. Specifications of two kinds were useful. Method contracts were used to specify that when a shared operation returns false no updates are made to the shared state and when it returns true changes are made only to the relevant parts. Object invariants were used to express that both the state before and after a method satisfy the object invariant.

Spec# translates contracts into a set of assertions and uses Boogie to statically verify these assertions. Boogie classifies assertions into provably correct assertions, provably failing assertions (these are flagged as warnings at compile time) and other assertions which cannot be proven statically. Spec# translates the last category of assertions into checks and throws up a warning if there is a violation at runtime. Programming with Spec# helped us in a few occasions to catch bugs in our application logic. For example, the Sudoku grid row check had an off by one error in array indexing which was caught with the aid of Spec#. For our final version of Sudoku with contracts, Spec# generated 323 assertions out of which boogie was able to verify 271 as correct while the remaining 52 were translated into runtime checks.

Maintaining local state. All updates to shared state happen via GUESSTIMATE and are internally protected by locks from within the GUESSTIMATE runtime. However, ensuring that updates to local state are protected by locks is the programmers responsibility. Particularly, as both completion operations and local operations can update local state, all state accessed within completion operations must be synchronized appropriately. For example, in the event planner application, both the local operations that mark tentative changes and the completion operations, update a `tentative_joined_list`. Care must be taken to protect this list with a lock.

In our experience, we find that the non-blocking nature of GUESSTIMATE is well suited to programming collaborative distributed applications. In the rare cases that blocking operations are required, they can implemented as shown in Figure 4.

7. Performance Evaluation

In this section we report the performance of the GUESSTIMATE runtime in terms of the time it takes for each synchronization (all three stages put together) to complete. We also study the scalability of applications written with guesstimate by measuring how increasing the number of users impacts (i) the synchronization time and (ii) the number of conflicts. All measurements were made while running the Sudoku application with 2 to 8 users within a local area network over a one hour time period. We chose to use Sudoku as the test application, since we could get several volunteers to use it with concurrent user activity.

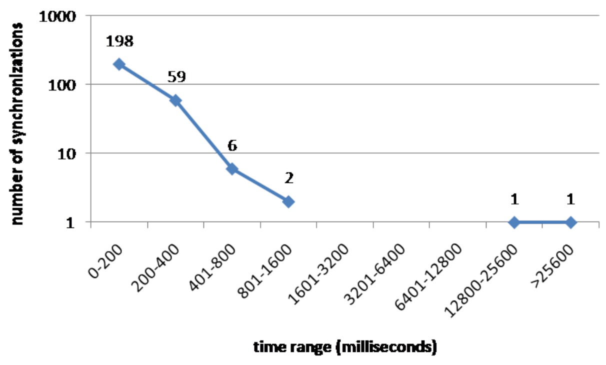

Figure 5. Distribution of time taken for synchronization

Synchronization time. Figure 5 plots the distribution of the time taken for synchronizations over a long run of the application involving 8 users solving 2 Sudoku grids. It can be seen that the time taken by guesstimate to complete a synchronization is within 0.5 seconds most of the time. There are 2 outliers in the distribution where a synchronization takes more than 12 seconds. These correspond to the times when synchronization stalled and the master had to perform a fault recovery. However, owing to the non-blocking nature of guesstimate even during these times the user could continue performing operations. Any blocking implementation would have rendered the system unresponsive at these times.

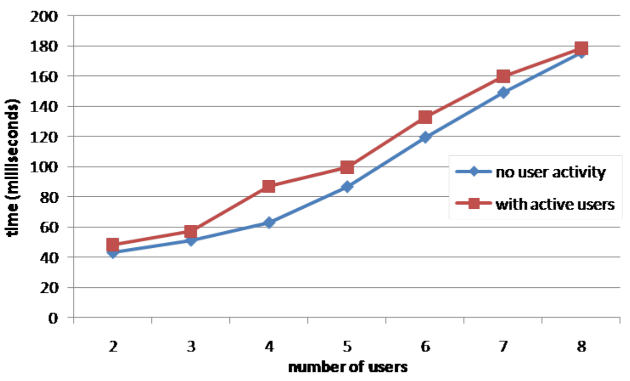

Figure 6. Average time to synchronize vs. number of users

Figure 6 shows the impact of number of users on synchronization time both in the presence and absence of user activity. The average synchronization time is measured by ignoring the outliers (time $>$ 12 seconds), as including them would skew the average away from the median. Two interesting observations can be made from this plot. First, the plot indicates that presence or absence of user activity does not affect the synchronization time by much. This indicates that the dominant component of the time for synchronization is network delay.

Second, it can be seen that the time for synchronization increases linearly with number of users. This can be attributed to the serial nature of the first stage (`AddUpdatesToMesh`) of synchronization (refer to Section 4). However, even assuming a linear increase guesstimate should easily scale to a 100 users as even with 100 users the average time to synchronize would be within 3 seconds. This is reasonable even for a high user activity event like a multi-player game, but possibly not for a highly sophisticated real time game. For collaborative applications which do not have such high user activity, guesstimate in its current form should scale to a 1000 users. To scale it further we would have to parallelize the first stage, which should also not be very hard. In our current design the first stage is kept serial purely for ease of monitoring and debugging.

Conflicts. Figure 7 shows the number of instances when an operation that succeeded on issue failed at commit time during our experiments. These measurements were made by adding a new user for every 100 synchronizations performed by the runtime. As can be seen conflicts are very rare even the presence of 8 active users.

Failure and recovery. During the one hour period for which we gathered statistics, GUESSTIMATE encountered three failures, once when one of the machines was restarted while the application was running, and twice when the synchronization was stalled possibly because a message was lost in transmission. GUESSTIMATE recovered in all three cases automatically, once by resending the lost message and twice by removing the machine from the stalled syn-

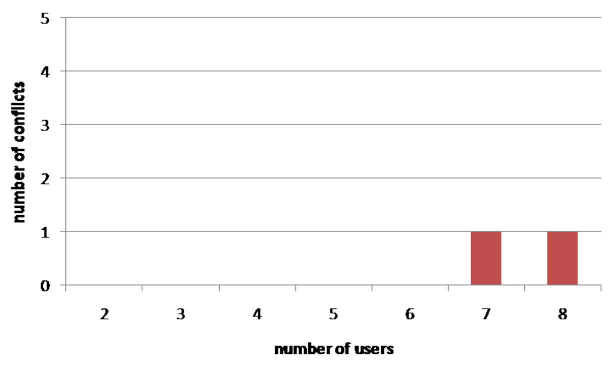

Figure 7. Number of conflicts vs. number of users

chronization loop and sending a restart message, and none of the other users were even aware of the failure.

8. Related Work

Distributed systems have existed for several decades now. Early research on consistency guarantees for data shared across distributed systems was from the database community [4]. The notions of replicated databases with one-copy serializability was introduced. One-copy serializability allows for pessimistic replication as the replicas are kept consistent at all times. These systems guarantee high consistency but can lead to low performance and availability [16]. Optimistic replication on the other hand allows for higher availability and faster performance than pessimistic replication. However, the replicas can diverge and safeguards need to be provided to ensure that some form of consistency is guaranteed. Systems with optimistic replication typically guarantee eventual consistency [14].

Many systems have been built for mobile and Internet services that support optimistic replication with an eventual consistency guarantee. The work most closely related to the guesstimate system is the mobile database system Bayou [12]. Bayou replicates a database on multiple servers and allows clients to submit SQL like queries to the database. This query is processed in one server and then propagated across to other servers using pairwise entropy. It allows for specification of per write merge functions to resolve conflicts. Bayou relies on features in the database storage system to log all operations on the database and to rollback and re-execute them multiple times so that all servers see operations in the same order. Little is known about the way to program applications for Bayou. GUESSTIMATE differs in that it provides a well defined API to build applications for distributed systems. It also does not need any guarantees from the underlying storage system. Unlike Bayou, GUESSTIMATE maintains all state in memory. In addition to **OrElse** operations that are similar to the merge functions defined in Bayou, GUESSTIMATE provides completion operations to notify application users about conflicts. Further, it allows users to naturally express related operations as **Atomic** operations.

The IceCube system [10] allows for users to specify constraints on the order in which operations should be executed. Constraints like either op_1 or op_2 should be performed and op_1 should only be performed if op_2 is performed can be specified. The final commit order can then be any order that satisfies all constraints. GUESSTIMATE respects the natural or causal ordering between operations submitted by the same user. In addition it allows for hierarchical operations.

Operational transformation [7] is a technique used for building collaborative applications that allows operations to be committed as soon as they are submitted. Operations received from other replicas are suitably transformed before being applied so that their semantic effect is preserved. While this is well suited for text applications where transformations can be carefully defined, writing such transformers is hard for other applications. Some of the design principles of GUESSTIMATE like replication, availability and conflict resolution are also used in the design of replicated and distributed file systems such as Unison [13], Coda [15] and GFS [8].

Mace is a language extension for C++ that allows the programmer to specify components of a distributed system in the form of a state transition model and transforms them into a C++ implementation. Mace also enables the use of code profilers and model checkers to help identify performance and correctness errors in distributed applications. GUESSTIMATE on the other hand provides specific support for the development of interactive applications and chooses a particular design point in the performance-consistency trade-off that is well suited to such applications. Other languages like X10 [6] target distributed systems with a partitioned global address space. Extensions to software transactional memory that scale to distributed systems with a partitioned global address space have also been proposed [5]. GUESSTIMATE is not restricted to any particular distributed system architecture.

In summary, the twin goals of providing a performance-consistency trade-off that is well suited to collaborative applications and a programming API that allows the programmer to build such applications sets GUESSTIMATE apart from existing work.

9. Limitations and Future Work

GUESSTIMATE has several limitations. We list them below together with some directions for future work.

Modularity and layering. Applications are often written in a layered fashion. While we have abundant experience using GUESSTIMATE to write applications in which the user directly interacts with the API via a user interface, we do not know if we can develop multi-layered software with it.

Size of shared state. GUESSTIMATE maintains multiple copies of the shared state and copies the committed state to the guesstimated state at the end of each synchronization. Dealing with large shared objects could therefore slow down guesstimate. However, optimizing copy-on-write has been studied in several contexts, such as STMs [1], and by using programming language features to inform the runtime about side-effect free code and optimize copying in such situations.

Updating local state. With GUESSTIMATE updating the local state whenever changes are made to shared state still remains the programmer's responsibility. Completion operations provide one way to update local state but these do not handle updates from remote operations. A mechanism to register a callback function for remote updates could prove useful.

Scalable run-time. As mentioned in Section 7 the current version of GUESSTIMATE might not scale beyond 1000 nodes, as the synchronization time increases linearly with number of users. While the non-blocking nature of GUESSTIMATE ensures that users are not blocked during synchronization, slow synchronization affects the lag between submission and completion, and hence affects user experience. One possibility is to parallelize the first stage of the synchronization protocol so that the time taken depends only on the number of operations and the network delay but not on the number of users.

Fault tolerance. While the current version of GUESSTIMATE tolerates some faults, since it is designed as a single master system, the master remains a single point of failure. One possibility is to extend the implementation to dynamically change the master in case the master fails.

Off-line updates. GUESSTIMATE does not currently support off-line updates. Adding such support could be non-trivial in the sense that if the time gap between the executions on the guesstimated and committed states is large, the scope for discrepancy and conflicts also becomes large.

ACKNOWLEDGMENTS

We thank Jim Larus, Dave Detlefs, Dave Campbell, Anders Hejlsberg, Rama Ramasubramanian, Sanjiva Prasad and the anonymous reviewers for their insightful comments on earlier drafts of this paper.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Programming language design and implementation*, 2006.

[2] Mike Barnett, K. Rustan M. Leino, K. Rustan, M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS '04: Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, 2004.

[3] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO '05: Formal Methods for Components and Objects*, 2006.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

[5] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08: Principles and practice of parallel programming*, 2008.

[6] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Object-oriented programming, systems, languages, and applications*, 2005.

[7] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD '89: SIGMOD international conference on Management of data*, 1989.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Symposium on Operating systems principles*, 2003.

[9] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[10] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Principles of distributed computing*, 2001.

[11] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9), 1979.

[12] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: replicated database services for world-wide applications. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, 1996.

[13] Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[14] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), 2005.

[15] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2), 2002.

[16] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *SOSP '01: Symposium on Operating systems principles*, 2001.