A Scalable Synchronous System for Robust Real Time Human-Machine Collaboration

Christopher Powers

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-87
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-87.html

May 28, 2020

Copyright © 2020, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I'd like to thank Dr. Fisher Yu for his mentorship on this project. Thanks to Professor Trevor Darrell, Professor Joseph Gonzalez, and my peers at Berkeley Deep Drive for their support. Finally, thanks to my family for everything.

A Scalable Synchronous System for Robust Real Time Human-Machine Collaboration

by

Chris Powers

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

EECS

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Trevor Darrell, Chair
Professor Joseph Gonzalez

Spring 2020

The thesis of Chris Powers, titled A Scalable Synchronous System for Robust Real Time Human-Machine Collaboration, is approved:

Chair ______________________ Date 5*28/20

______________________ Date May 28, 2020

______________________ Date ______

University of California, Berkeley

A Scalable Synchronous System for Robust Real Time Human-Machine Collaboration

Copyright 2020
by
Chris Powers

Abstract

A Scalable Synchronous System for Robust Real Time Human-Machine Collaboration

by

Chris Powers

Masters of Science in EECS

University of California, Berkeley

Professor Trevor Darrell, Chair

Large labeled datasets are vital for applying supervised learning to computer vision tasks, but making these datasets takes time and effort. Human-machine collaboration has the potential to mitigate this cost. This work introduces a general-purpose framework for human-to-human and human-machine collaboration on image data. We show that by treating machine learning models as virtual users, multi-user synchronization can support versatile human-machine interaction; in other words, all you need is sync. In order to achieve synchronization behavior that seems correct to users, while also maintaining real-time editing speeds and supporting undo-redo, we adapt operational transformation [8] to the image labeling context. An open-source implementation of the collaboration system is presented as an in-progress addition to Scalabel, an annotation tool for visual data that was used to create the BDD100K dataset [24]. Finally, we give an example of how the collaboration feature can improve the labeling process by integrating Polygon-RNN++ [2] with Scalabel.

Contents

Contents **i**

List of Figures **ii**

1 Introduction **1**

2 System Design **2**
2.1 Motivation 2
2.2 System Overview 3
2.3 Autosave 7

3 Synchronization **12**
3.1 Background 12
3.2 Problem Description 13
3.3 Operational Transformation 16
3.4 Transformation Rules for Scalabel 18
3.5 Undo-Redo 20
3.6 Model Serving Example 23
3.7 Future Work 23

References **25**

List of Figures

2.1 GUI for Scalabel. The type of the annotation task, 2D bounding box, is shown in the upper-left. Users can draw a box then edit its category or attributes with the menu on the left. They can also navigate through a sequence of items, which might represent a series of images from a video. Finally, the save button appears when autosave is disabled. . . . 3

2.2 The system architecture for human labeling. The nodes are split between blue for web clients and green for server-side components. . . . 4

2.3 The flow of information during multi-user collaboration. When a user takes an action, it is sent to the server for synchronization, after which it is broadcast to all other users collaborating on the same task. . . . 5

2.4 The full system architecture. The new orange nodes are the components that involve machine-learning models. Bot users and web users both interact with the server hub through the same action synchronization interface. . . . 6

2.5 The robust autosave algorithm. The client and server interface with each other via the **socket** connection. The autosave flow starts when a client runs `ExecuteAction`, which triggers `HandleAction` in the server. The server uses `AtomicSave` to save the result of the action, then triggers `ConfirmAction` on the client to complete the process. For sync, `ConfirmAction` also executes broadcasted actions from other users. In contrast to the client, which has a local copy of **state**, the server is stateless and reads all of its data from redis. . . . 9

2.6 The robust registration algorithm. First, a client runs `StartRegister`, which triggers `Register` in the server. Next, the server sends its copy of the state to `FinishRegister`. Once the client is up to date, it resends any unconfirmed actions using the `HandleAction` method defined in Algorithm 2.5. . . . 10

2.7 Finite state machine for the web user. Nodes are split into green, where all local data is saved, blue, where there is unsaved local data, and orange, where a warning dialogue is shown. . . . 11

3.1 A failure case for the naive synchronization algorithm. The consistency requirements fail regardless of action parameterization. Fixing this issue requires introducing OT to the system. . . . 14

3.2 Fixing the naive algorithm's failure cases via operational transformation. User 2 executes the modified action A_1^* instead of the original broadcasted action A_1. . . 15

3.3 The client-side algorithm for transforming out-of-order actions. The list inclusion transformation (`LT`) from [21] includes the effects of A_m to A_n into A, so that applying A' should produce the correct state. To prevent $\mathbf{H}$ from growing indefinitely, it is periodically cleared in periods of low-activity, which can be identified via distributed-consensus [8]. 17
3.4 JSON structure of a task with one item. Only a minimal subset of relevant properties are shown. Boxes are objects with string or numeric properties, like category name or position coordinates. The ordering list tracks which labels should be displayed in front/behind others, which is useful for marking occluded objects. 19
3.5 The final undo/redo algorithm. Only actions that are originated by the local user are added to the undo stack. Undo and redo are high level functions locally, but ultimately execute normal actions which can be synchronized between users. The operational-transformation components are left out for simplicity. 22
3.6 Polygon-RNN++ predictions synchronized to the Scalabel web user. 24

Acknowledgments

I'd like to thank Dr. Fisher Yu for his mentorship on this project. Thanks to Professor Trevor Darrell, Professor Joseph Gonzalez, and my peers at Berkeley Deep Drive for their support. Finally, thanks to my family for everything.

Chapter 1

Introduction

Large labeled datasets such as ImageNet [7], Coco [14], and BDD100K [24] are vital for applying supervised learning to computer vision tasks, but making these datasets takes time and effort. Human-machine collaboration has the potential to mitigate this cost. Increasing user involvement in the learning processes [3, 5, 12], and conversely increasing machine involvement in the labeling process [6, 19, 25], can have a positive effect. Interactive machine learning [9] increases the frequency of learner updates, which makes more room for humans in the learning process. For example, Polygon-RNN++ [2] allows users to shift the vertices of its predicted polygon labels, then uses these corrections as feedback for improvement.

This work introduces a general-purpose framework for human-machine collaboration on image data. It also contributes an in-progress implementation of the framework as part of Scalabel[1], an open-source annotation tool for visual data that was used to create the BDD100K dataset [24]. With the framework in place, Scalabel will be able to support collaboration with a wide range of interactive models.

Chapter 2 outlines the system design for Scalabel in the context of supporting collaboration between multiple humans as well as between humans and machines. First it explains the single user architecture, then it shows how to support multi-user collaboration via synchronization. The key insight, all you need is sync, is that synchronization also serves as a flexible medium for human-machine interaction. Under this paradigm, models are represented as virtual users that act similarly to human ones. Finally, it takes a step towards a robust synchronization algorithm by detailing the procedure for automatic saving (autosave).

Chapter 3 defines the synchronization algorithm for Scalabel. In order to achieve correct synchronization behavior while maintaining real-time editing speeds, it adapts the general technique of operational transformation (OT) [8] to the specific image labeling domain. It also explains how the OT-based algorithm can support undo and redo operations with minimal additional overhead. Finally, it gives an example of human-machine collaboration on Scalabel with Polygon-RNN++ [2].

[1]Can be accessed at `https://github.com/scalabel/scalabel`

Chapter 2

System Design

2.1 Motivation

Effective human-machine collaboration has huge value for supervised-learning based computer vision tasks. Active learning [6] improves human labeling efficiency by leveraging the model's confidence scores to select the next batch of data to label. Similarly, the LSUN dataset [25] extracts extra value per human label by repeatedly alternating between human labeling and model training steps. Similar methods [19] alternate between soliciting user feedback and updating the model prediction probabilities. By increasing labeling efficiency, these techniques allow the creation of larger, more accurate, and more diverse datasets.

Interactive models [9] can increase the frequency of user feedback to the model from the level of a batch to the level of an individual prediction. For example, Polygon-RNN++ [2] allows users to shift the vertices of its predicted polygon labels, then uses these corrections as feedback for improvement.

Moving to other contexts, power to the people [3] studies the benefits of letting users closely interact with models. Combining human and machine inputs helps solve crowdsourcing tasks [12]. Human interaction with computers combines their relative strengths; for instance, users can identify low-level visual features to guide the model [5] .

In contrast to human-machine collaboration, real time collaboration between humans on image data is less studied, but has several applications, such as letting an expert annotator guide a novice, or letting one annotator provide quality assurance for another.

This work adds support for both human-to-human and human-machine collaboration to the Scalabel annotation tool. This enables interaction with models for 2D bounding box, polygon segmentation, 2D tracking, and 3D bounding box tasks, since Scalabel has all of these labeling modes. The system supports a large number of users, since Scalabel is a web application designed for scalability. Figure 2.1 shows the graphical user interface (GUI) for 2D bounding box annotation.

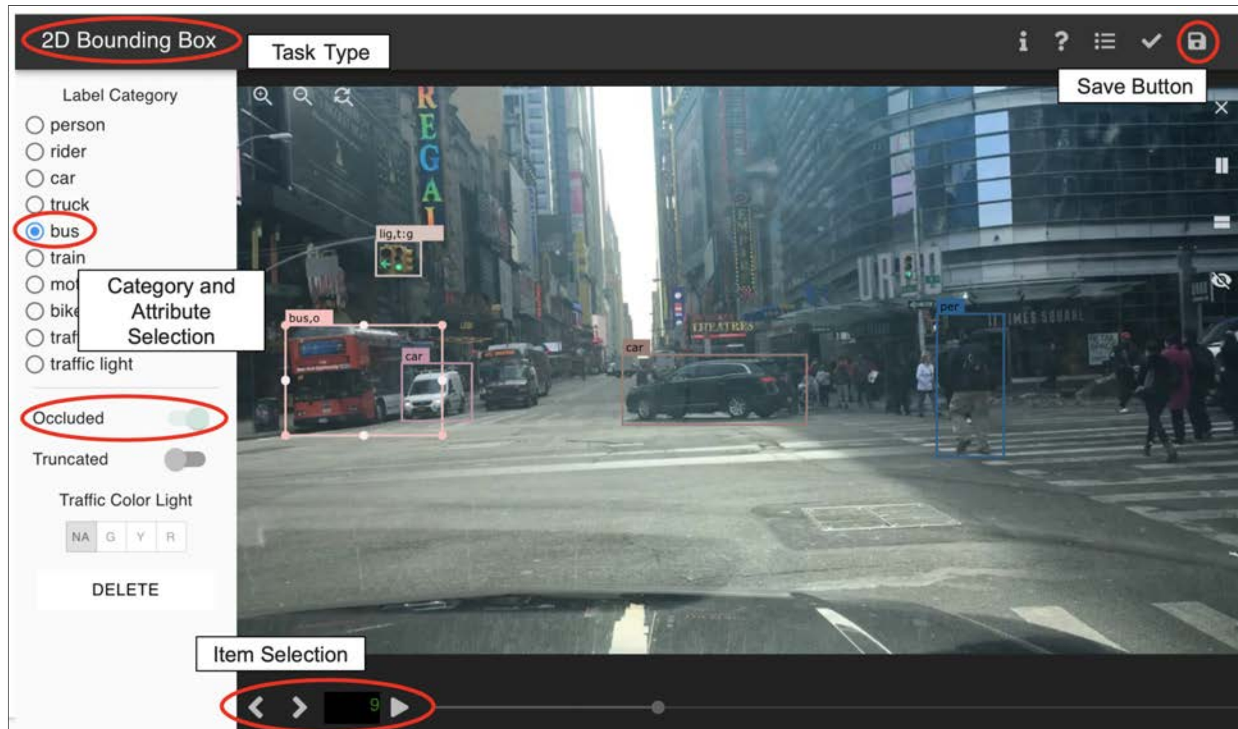


Figure 2.1: GUI for Scalabel. The type of the annotation task, 2D bounding box, is shown in the upper-left. Users can draw a box then edit its category or attributes with the menu on the left. They can also navigate through a sequence of items, which might represent a series of images from a video. Finally, the save button appears when autosave is disabled.

2.2 System Overview

Architecture

Scalabel aims to support an arbitrary number of user simultaneously labeling image data. Even without collaboration, different users must be able work on different sets of images, called tasks, without any synchronization. Moreover, the interface for connecting machine learning models to the system should be general enough to support a wide range of interaction. Thus, the system design for collaboration and synchronization must be scalable, flexible, and robust.

The architecture for single user labeling and human-to-human collaboration is shown in Figure 2.2. It supports two types of communication between client and server. HTTP is used for infrequent requests such as creating or exporting a project. This paper focuses on the second type of communication, which uses Socket.IO to synchronize user actions, represented by their parameters, between the client and server. For example, after the user draws a bounding box, the client sends the server a message containing the box's positional

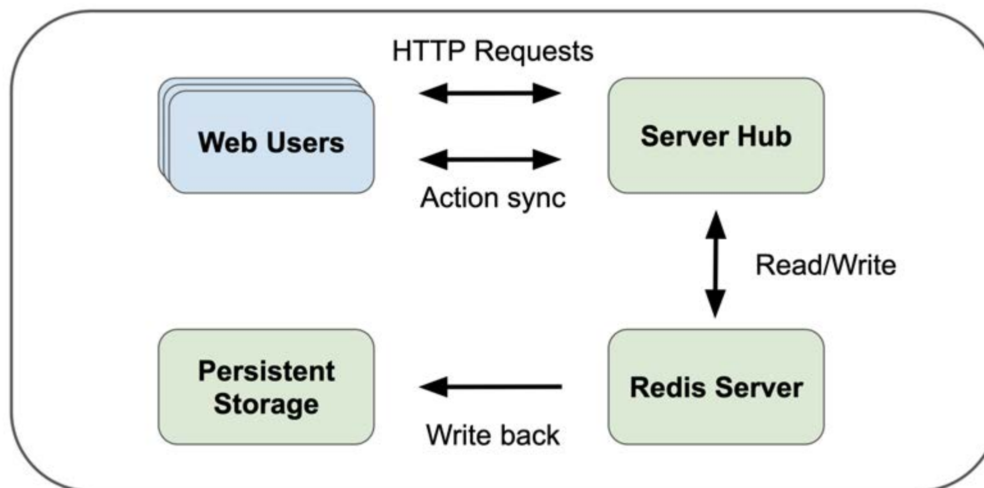


Figure 2.2: The system architecture for human labeling. The nodes are split between blue for web clients and green for server-side components.

coordinates. The server uses the received action to update its copy of the client task's state; for example, it adds the new box's information to the list of labels for an image.

Implementing this system requires a carefully defined structure for actions and state, as well as a set of rules for how actions update state. For client and server state to match, they must use the same rules and structures, which is easiest if they share the same code. To this end, we used a TypeScript frontend and NodeJS backend which share the Redux-based implementation of the action-state system.

Alternative designs do not require the server to interact with state; for instance, if the client sends a serialized copy of the entire state, the server can just save and load it. We choose the action based approach because actions contain much less information than the state, so sending them is more bandwidth efficient, especially for the high traffic of autosave. Furthermore, during collaboration, the server state can serve as a master copy to synchronize the different users.

Multi-user Collaboration

Human-to-human collaboration is the natural first extension to the single user system. It allows users to label the same task and even the same image simultaneously, which can be useful for training new annotators or providing real time quality assurance. Figure 2.3 shows how collaboration occurs on the existing architecture. Unlike the single user system, which just sends a confirmation message to the original user, the multi-user system keeps clients in sync by broadcasting every action update it receives to all users working on the task. Chapter 3, Synchronization, explains how to ensure correct and consistent behavior under

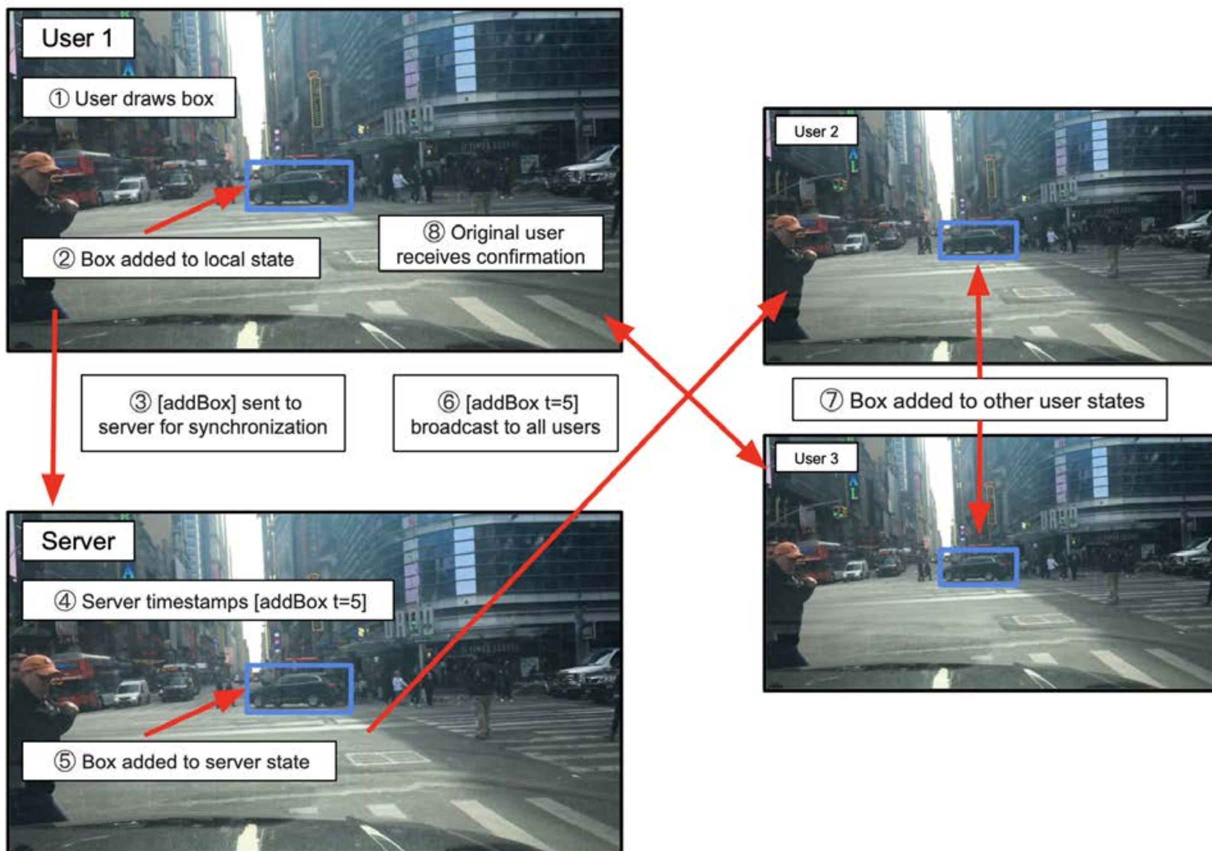


Figure 2.3: The flow of information during multi-user collaboration. When a user takes an action, it is sent to the server for synchronization, after which it is broadcast to all other users collaborating on the same task.

this model in the face of challenges such as lag and conflicting user intentions.

All you need is sync

With the architecture for multi-user collaboration in place, the next step is collaboration between humans and machines. Ideally, collaborative interaction could take many forms, such as humans refining the predictions of a model, models using human input to make better predictions, or humans and machines iteratively providing feedback to each other. Our key insight is that the existing synchronization infrastructure can also support synchronization with models. The expanded system diagram in Figure 2.4 introduces bot users, which interact with the server by sending actions over a Socket.IO connection, just like web users. Since bot users can take any action available to a regular user, they have maximum flexibility and are not restricted to a particular mode of interaction. The phrase "All you need is sync"

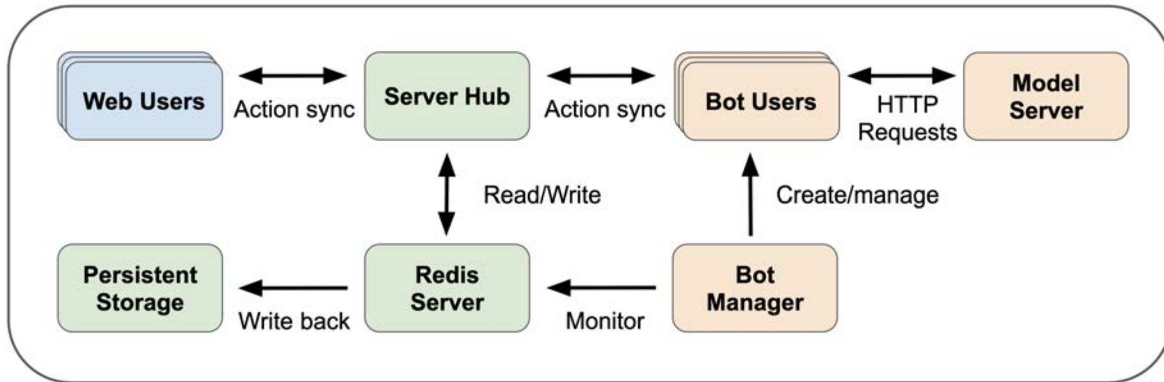


Figure 2.4: The full system architecture. The new orange nodes are the components that involve machine-learning models. Bot users and web users both interact with the server hub through the same action synchronization interface.

summarizes this paradigm of using synchronization as a versatile medium for collaboration.

Bot users interact with the server in the same way as web users, and use the same NodeJS state-action management code, but they differ on other fronts. Web users are initialized by actual humans, whereas bots are initialized by a bot manager process. Moreover, the actions of web users are generated by actual humans, but the actions of bot users must come from models, which live on a separate server with an HTTP API.

The bot manager handles creation and deletion of bot users. Whenever the server registers a new connection, it publishes a registration event to Redis. By subscribing to these events, the bot manager creates a new bot for every active task. In addition, it deletes bots for inactive tasks by monitoring how many broadcasted actions every bot user receives.

The model server performs the actual model computations. When a bot user receives a broadcasted action, it sends an HTTP request, called a query, to the model server under two conditions:

1. The action must have model support; for example, if there is no 3D bounding box model, nothing will happen during 3D scene annotation.

2. The action must be made by a human, not another bot, to prevent infinite recursion of predictions. This is determined by marking bot actions with an *isBot* field.

To allow a variety of requests, queries are specified in the flexible BDD100K data format, which contains an image along with any associated 2D box, 3D box, or 2D polygon labels. The model server in Python receives the query, reformats it to match the model input, then returns the prediction to the bot user. If a user takes multiple actions before saving, the corresponding queries will be bundled together, allowing the model to run its prediction in batch mode. The bot user executes the action-equivalent of the prediction, which broadcasts to every user.

Scalability

Scalabel should support a large number of users working simultaneously, so each component must be able to scale up as needed. The core scalability goals are to eliminate bottlenecks at each stage of the pipeline and to make the system flexible and easy to extend.

The first bottleneck is the server hub, which updates the master copy of the state for every task. Scaling up means load balancing requests to multiple copies of the server code running in parallel. Synchronizing state between each server process would be challenging; instead, all server processes are stateless. The Redis server acts as a distributed key-value cache for all data, including task states, bot manager information, and metadata. Server processes read data from Redis, perform some operations, then save the data back to Redis, all in one atomic transaction to ensure consistency. In addition, Redis writes back to persistent storage after a specified number of actions, or after a specified period of inactivity, whichever comes first. Since local file storage is impractical for a distributed system, Amazon S3 storage is supported.

The other main bottleneck is the model server, which manages the computations for a variety of different models and task types; for example, it may have to simultaneously serve predictions of bounding boxes, polygon instance segmentations, and 2D tracks. Scaling up the model server occurs independently from the rest of the system, since the only communication is HTTP. One possible solution is Ray [16], which can autoscale a distributed network of machine learning models based on system load. Managing the allocation of GPU, CPU, and memory resources is also an interesting problem for future work.

Lastly, the interface for adding new models to the system must be as streamlined as possible. The generality of the bot user API and the BDD100K query format puts the current bottleneck at the conversion interface between action format and model input/output format.

2.3 Autosave

Adding autosave

Autosave is expected in most modern web applications, and is doubly important for collaborative applications because they require constant synchronization of state. Thus, we add autosave to the system as a convenient feature for single-user labeling and a near-necessary component for any human-human or human-machine collaboration efforts.

With autosave enabled, actions are synchronized to the backend whenever they are executed, instead of waiting for the user to click the save button. In both autosave and manual saving, some types of actions are exempt from synchronization:

- Actions that affect user-specific state, like the currently selected image, are only executed locally.

- Actions received via a broadcast from another user are only executed locally in order to prevent them from repeatedly synchronizing with the server. To track this, actions are marked with a *userID* field identifying the web client where they originated.

Robustness

The basic procedure for saving is simple, but failures with the server, client, or connection can cause data loss. To solve these issues, we develop robust algorithms for autosaving (Figure 2.5) and user registration (Figure 2.6). Additionally, we provide useful feedback to the web user by tracking their status with a finite state machine (FSM) model.

To achieve robustness, every user action must eventually be executed on the server state exactly once. For multi-user collaboration, actions must also be executed by all other users on the same task exactly once. In some cases, data loss is inevitable; for instance, if both server and client crash, any unsaved progress will be lost. The algorithms handle cases where the server or connection fails, but the client stays online. Unnecessary data loss in other cases is minimized by the guidance provided by the FSM.

Algorithm 2.5 shows how the basic autosave procedure is augmented to increase its robustness to errors. The closely related Algorithm 2.6 shows how clients register with the server, whether to connect for the first time or to reconnect after a prior failure. Key additions are:

1. The web client tracks actions sent to the server, in case it needs to resend them.
2. When it reconnects, the web client synchronizes with the server state, then re-applies and resends all the sent but unconfirmed actions.
3. The server tracks the IDs and timestamps of actions that have been saved to ensure that each action is saved exactly once.
4. The server updates the task state and the saved IDs atomically, meaning both will be saved to redis, or neither will.

Many of these additions also help make manual saving more robust, with some slight modifications such as sending a queue of unsaved actions instead of a single action at a time.

Server-side crashes can occur if the server process is accidentally stopped, if the machine hosting the server fails, or if some unexpected error occurs. The autosave algorithm prevents data loss in these scenarios, which can be classified into two types with respect to a given action. First, the server can crash before the updated state is saved to redis in line 13 of Algorithm 2.5. Second, the server can crash after saving on line 13, but before sending the confirmation on line 7. In either case, the server-side crash triggers a reconnection event, so the client resends all the unconfirmed actions, as described in Algorithm 2.6. In the first case, the server has no record of ever receiving the actions before, so it saves as normal. In the second case, the server already saved the updated state, along with the action ID and

```
: Web Client
   state: the local task state
   socket: connection to the server
   sentActions: maps ID to sent action
   confirmedActions: confirmed IDs
   userID: user identifier

   Def ExecuteAction(A)
1     state.update(A)
2     if A.userID = userID then
3        A.ID = GenID()
4        sentActions[A.ID] = A
5        socket.HandleAction (A)

   Def ConfirmAction(A)
6     sentActions.remove(A.ID)
7     confirmedActions.add(A.ID)
8     if A.userID ≠ userID and
9     A.ID ∉ confirmedActions then
10       ExecuteAction(A)
```

```
: Server
   redis: the redis client
   socket: connection to the web client

   Def HandleAction(A)
1     savedIDs = redis.GetSavedIDs ()
2     if A.ID ∉ savedIDs then
3        AtomicSave(A)
4     else
5        A.timestamp = savedIDs[A.ID]
6     end
7     socket.ConfirmAction(A)

   Def AtomicSave(A)
8     state = redis.GetState (A.taskID)
9     state.update(A)
10    A.timestamp = GetTime()
11    savedIDs = redis.GetSavedIDs()
12    saveIDs[A.ID] = A.timestamp
13    redis.AtomicWrite (state,
       savedIDs)
```

Figure 2.5: The robust autosave algorithm. The client and server interface with each other via the **socket** connection. The autosave flow starts when a client runs `ExecuteAction`, which triggers `HandleAction` in the server. The server uses `AtomicSave` to save the result of the action, then triggers `ConfirmAction` on the client to complete the process. For sync, `ConfirmAction` also executes broadcasted actions from other users. In contrast to the client, which has a local copy of **state**, the server is stateless and reads all of its data from redis.

timestamp, to Redis in an atomic operation. Thus, the server handles the repeated action by re-applying the old timestamp then sending confirmation to the client.

Failure during multi-user collaboration, such as a client going out of sync with the server, can cause unpredictable behavior. This could occur if a client never receives an action, or if it executes a broadcasted action too many times. For the first case, since the server simultaneously sends the confirmation and broadcast, the existing confirmation protocol ensures that the broadcast is always sent. To prevent the second case of executing an action too many times, each web user tracks the IDs of confirmed actions and does not re-execute them.

Finally, some failures are unavoidable, but can be mitigated by appropriately guiding the user. First, long periods of disconnection can occur for users with unstable internet. To prevent these users from diverging too far from the server state, their drawing canvas

```
: Web Client
  state: the local task state
  socket: connection to the server
  sentActions: maps ID to sent action
  taskID: task identifier

  Def StartRegister()
1 |   socket.Register (taskID)

  Def FinishRegister(serverState)
2 |   state = serverState
3 |   for action A in sentActions do
4 |   |   state.update(A)
5 |   |   socket.HandleAction (A)
6 |   end
```

```
: Server
  redis: the redis client
  socket: connection to the web client

  Def Register(taskID)
1 |   state = redis.GetState(taskID)
2 |   socket.FinishRegister(state)
```

Figure 2.6: The robust registration algorithm. First, a client runs `StartRegister`, which triggers `Register` in the server. Next, the server sends its copy of the state to `FinishRegister`. Once the client is up to date, it resends any unconfirmed actions using the `HandleAction` method defined in Algorithm 2.5.

is frozen to prevent further edits, and they are notified that the client is attempting to reconnect. Once the connection is reestablished, the registration algorithm resynchronizes the user. Second, simultaneous failures on client and server inevitably result in the loss of unsaved data. To alleviate this problem, users are warned whenever they have unsaved data and attempt to exit or refresh the labeling interface. The finite state machine in Figure 2.7 illustrates how the connection status of the client is tracked. Besides enabling the frozen canvas and warning features, the status is also used to notify users when saving succeeds.

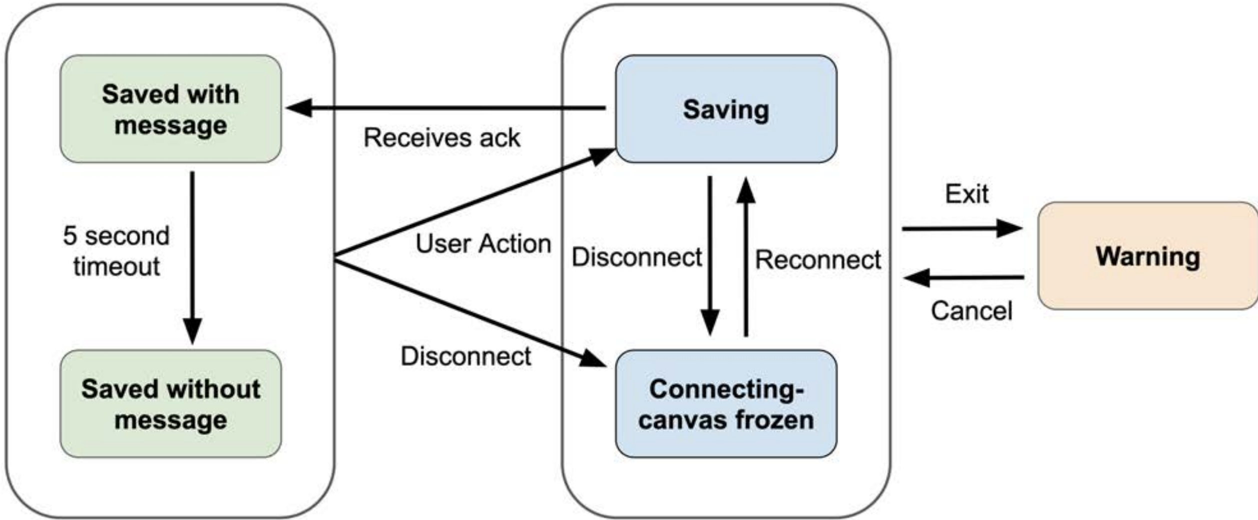


Figure 2.7: Finite state machine for the web user. Nodes are split into green, where all local data is saved, blue, where there is unsaved local data, and orange, where a warning dialogue is shown.

Chapter 3

Synchronization

3.1 Background

Robust synchronization is a requirement for the human-to-human and human-machine collaboration architecture discussed in Chapter 1. Collaborating users are in-sync if they see the same set of label data with the same category/attributes, i.e. if they have the same state. Keeping users in-sync is challenging because of connection problems like lag, inherent problems like conflicting user intentions, and the fact that any error can compound, since future edits on a divergent state may be applied incorrectly. This chapter introduces a general framework for multi-user synchronization, then adapts it to Scalabel and the image labeling context. With this framework in place, an arbitrary number of Scalabel users can reliably collaborate with each other or with models while maintaining real-time editing speed and all labeling functionality, including undo-redo. An example use case of collaborating with Polygon-RNN++ [2] is provided.

Operational transformation (OT) is a general purpose synchronization framework introduced in [8]. Its goal is to ensure all users have consistent state, but it recognizes that users may receive and have to execute broadcasted actions in different orders. To solve this problem, users transform the broadcasted actions, or operations, to achieve the correct effect on their own state. OT can support collaborative versions of MS Word and Powerpoint [22], and can serve as a basis for P2P collaborative algorithms like WOOT [17]. OT guarantees several properties, including convergence of state, preservation of user intention, and preservation of the causal ordering of events [10, 18, 21]. Scalabel's centralized architecture substantially simplifies the implementation of OT-based synchronization.

Alternative approaches include three-way merges, differential synchronization, and CRDTs. Three-way merges [15] compare and merge states instead of synchronizing differential edits, but they cannot easily support real-time collaboration. Differential synchronization [11] builds on this idea by using continuous diff and patch operations to keep client and server in sync. For Scalabel, an edit-based sync is easier to implement because all user actions are already explicitly defined for Redux. Conflict-free replicated data types (CRDTs) [20] allow

distributed replicas of the state to self-stabilize and eventually achieve consistency. The state structure for Scalabel is designed to minimize conflicts, but for cases where conflict is unavoidable, OT is a necessary supplement.

Undo-redo in multi-user systems is a well studied problem [1, 4, 10, 21]. Defining the expected behavior of undo at a high level is challenging because each user provides a separate flow of control [1]. Nevertheless, OT can be used to support undoing any operation at any time [21]. Our system implements a simple undo-redo mechanism for undoing the most recent local action.

3.2 Problem Description

Naively using the existing synchronization architecture can result in unintended behavior or divergent state between users. Recall that a single server coordinates multiple clients, as opposed to a peer to peer system where multiple clients could coordinate via vector clocks [13]. During synchronization, actions flow from the originating client to the server, where they are broadcasted to all other clients for the task, as well as to the original client for confirmation. To maintain real-time responsiveness, actions are executed locally before they are confirmed by the server. This technique, called input prediction [23], is a common feature of systems where OT can help [8].

Input prediction causes problems with synchronization, as shown in Figure 3.1. Even in this simple example of two clients trying to edit the same label, errors occur regardless of action parameterization; however, the two errors are different in nature. In Figure 3.1a, all states are in-sync, but the final result is not what either user intended. In Figure 3.1b, the server state matches the users' intentions, but one client is out-of-sync.

The different types of errors can be formalized using the OT consistency model [10, 18, 21], which requires three properties to hold for any consistent collaborative editor:

1. **Intention Preservation**: the effect of executing an operation in any state should be the same as its intention, the effect it would have had in the state where it was generated. This corresponds to the first type of error.

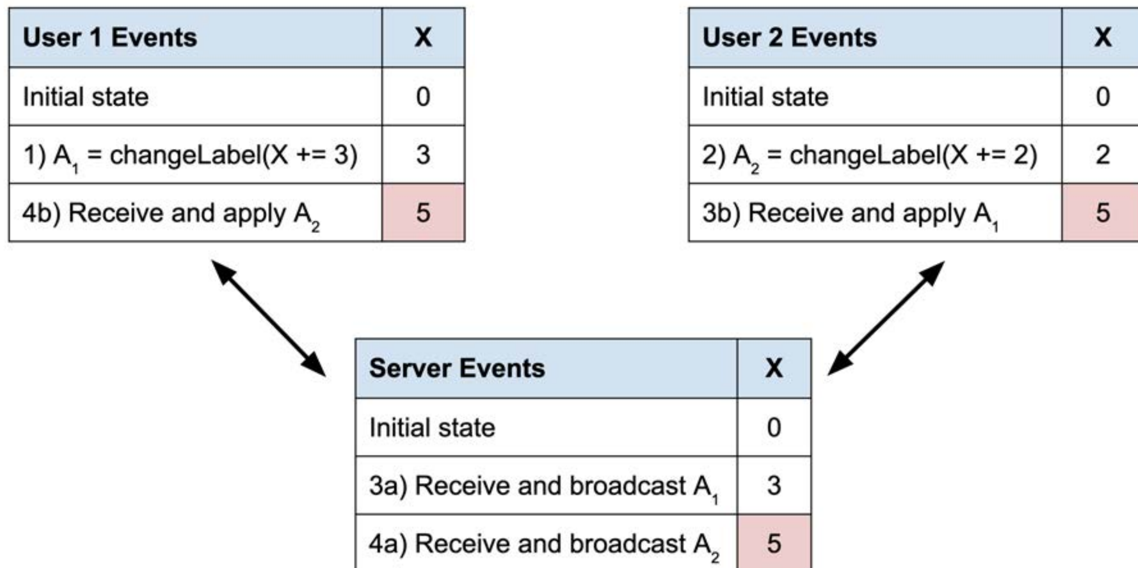
2. **Convergence**: also known as eventual consistency. After all nodes have executed all operations, the states should be identical. This corresponds to the second type of error.

3. **Causality Preservation**: the final state should have A executed before B, or appear as if A executed before B, if A happened-before B (A $\rightarrow$ B). Per the definition in [13], A $\rightarrow$ B if one of the following holds:

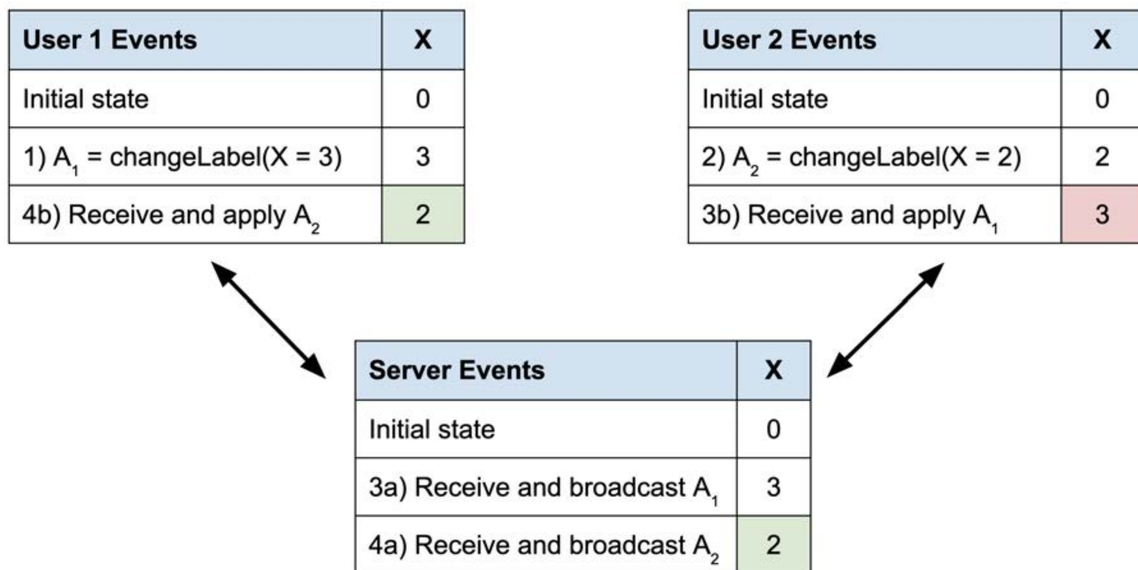
    a) A and B are on the same node, and A was generated before B.

    b) Sending A from one node triggers a response B in another node.

    c) Transitivity: A $\rightarrow$ B and B $\rightarrow$ C implies A $\rightarrow$ C.



(a) Two users simultaneously try to increment the X coordinate of the same label, which makes the X coordinate much larger than either user intended.



(b) Two users simultaneously try to reassign the X coordinate of the same label. Due to input prediction, User 2 executes the actions in the wrong order, and since the actions are not commutative, User 2 goes out of sync.

Figure 3.1: A failure case for the naive synchronization algorithm. The consistency requirements fail regardless of action parameterization. Fixing this issue requires introducing OT to the system.

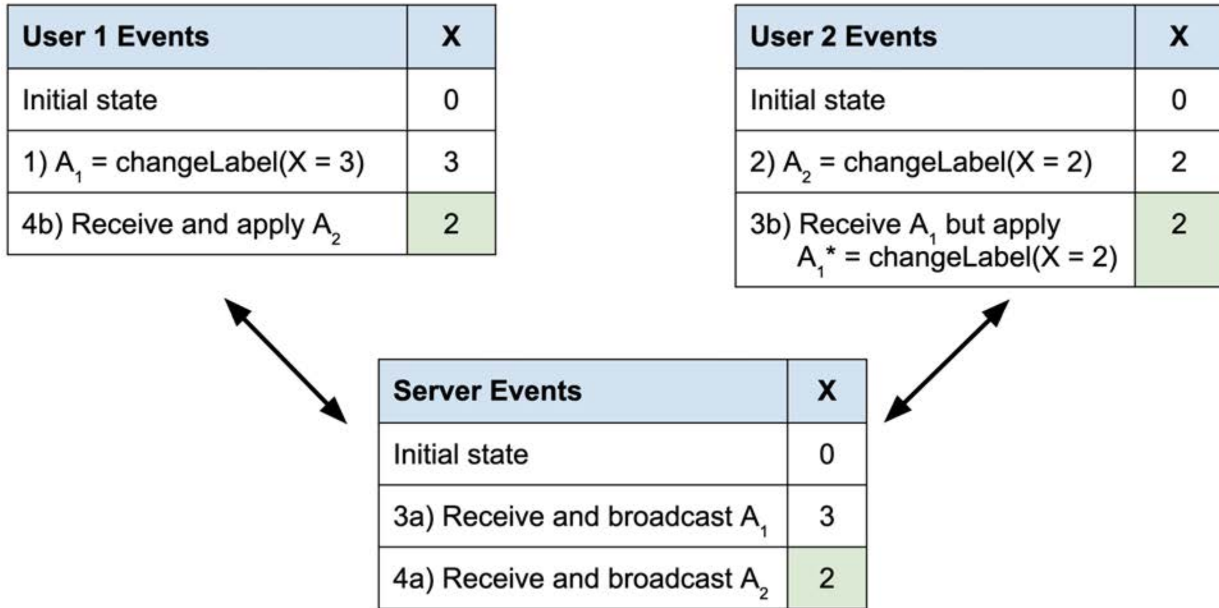


Figure 3.2: Fixing the naive algorithm's failure cases via operational transformation. User 2 executes the modified action A_1^* instead of the original broadcasted action A_1.

Causality preservation is a natural consequence of Scalabel's client-server architecture. Actions are generated by web users, but aren't assigned timestamps until they reach the server. The OT algorithm discussed later uses these timestamps to decide the order of action execution. This timestamp-based ordering satisfies the happened-before relationship. First, two actions from the same user are sent to the server in the order they are generated, so they receive timestamps in that order. Next, two actions from different users have no inherent causal relationship; thus, whichever arrives at the server first receives an earlier timestamp since it happened-before the other. This strategy of ignoring the physical time of events in favor of the time they arrive at the server is a necessity for handling poor connections.

Preserving the intention of users is important because the system behavior should not be drastically different with or without collaboration. In Figure 3.1b, parameterizing changeLabel by the new X value, instead of by the shift in X, naturally preserves intention. Carefully designing the action parameterization also works for other actions. For example, consider the scenario where labels are stored in a list, and are deleted by their index in the list. If two users both try to delete the label at index 0, the server will delete the labels at indexes 0 and 1, which neither user intended. Specifying label deletion via a unique label ID, instead of by list index, fixes the problem by making the extra deletion do nothing.

Convergence is vital for multi-user systems, since otherwise state data becomes corrupted and collaboration becomes impossible. Moreover, the previous fixes for intention preservation often make actions non-commutative, which breaks convergence in the naive model. This

is where operational transformation helps. Figure 3.2 shows how the previous convergence issues can be resolved by transforming the broadcasted action before executing it. The next section explains how the correct transformation is chosen, and under what circumstances a transformation is applied.

3.3 Operational Transformation

Notation

Some notation is necessary to succintly describe the OT procedure:

- Actions are functions that map an old state to a new one: $S' = A(S)$, or just $S' = A*S$.
- Actions can be composed: $(A * B)(S) = A(B(S))$.
- Actions are commutative if $A * B = B * A$.
- The transformation function maps a pair of actions to a new action: $A' = T(A, B)$.
- A list of actions H = $[A_1 \ldots A_n]$ can be applied to the state:

$$\begin{aligned} H * S &= H[n-1] * \ldots * H[0] * S \\ &= A_n * A_{n-1} * \ldots * A_1 * S \end{aligned}$$

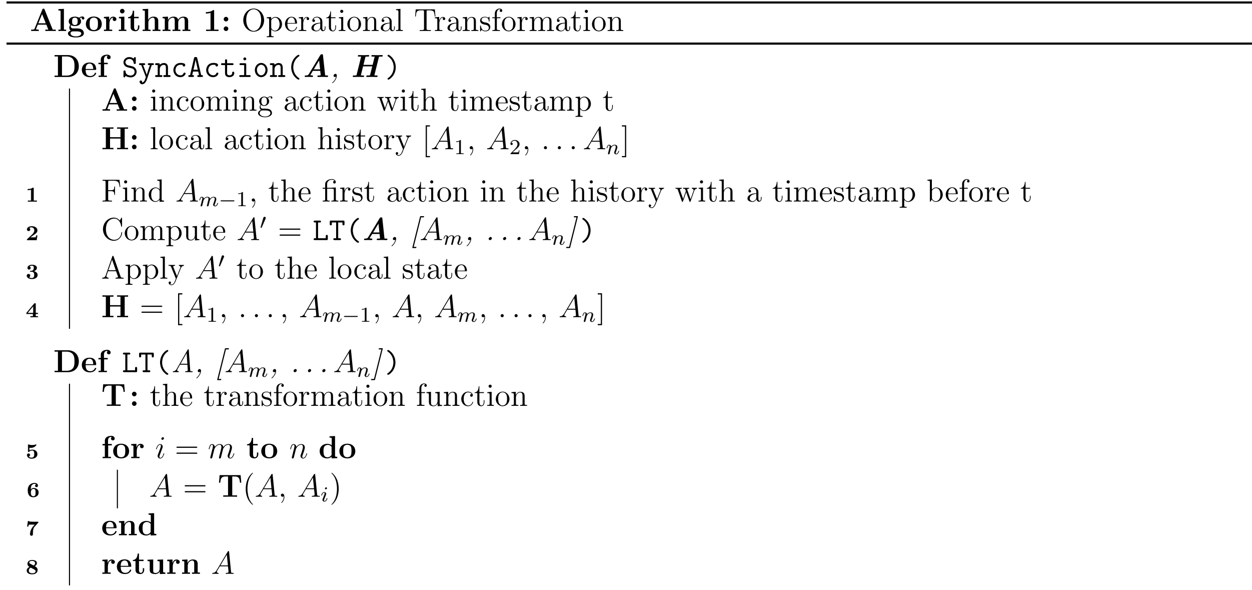
- A list of actions can be spliced: H[j:k] = [H[j], H[j+1], . . . , H[k-1]]

The Transformation Function

Defining the correct transformation function is the key to fixing the convergence problems for synchronization. Say the server receives action A before B, but a user executes B first due to input prediction. To ensure the user's state remains correct, action A must be transformed before being executed. The specific transformation depends on action B, so the modified action A' is called the result of transforming A against B, or $A' = T(A, B)$. After the transformation, the server state $B(A(S))$ must match the user state $T(A, B)(B(S))$. More concisely, the condition for correctness is:

$$T(A, B) * B = B * A$$

If A and B are commutative, $T(A, B) = A$ satisfies this condition. This motivates making as many action pairs commutative as possible in order to reduce the number of transformation rules. In the worst case, a transformation matrix of n^2 rules is required to describe how every action transforms against every other [8]. By parameterizing actions in terms of pointers, [1] reduces this to n rules for how each action affects the pointers.



Algorithm 1: Operational Transformation

Def SyncAction(***A***, ***H***)
 A: incoming action with timestamp t
 H: local action history $[A_1, A_2, \ldots A_n]$
1 Find A_{m-1}, the first action in the history with a timestamp before t
2 Compute $A' =$ LT(***A***, $[A_m, \ldots A_n]$)
3 Apply A' to the local state
4 $\mathbf{H} = [A_1, \ldots, A_{m-1}, A, A_m, \ldots, A_n]$

Def LT(A, $[A_m, \ldots A_n]$)
 T: the transformation function
5 **for** $i = m$ **to** n **do**
6 $A = \mathbf{T}(A, A_i)$
7 **end**
8 **return** A

Figure 3.3: The client-side algorithm for transforming out-of-order actions. The list inclusion transformation (LT) from [21] includes the effects of A_m to A_n into A, so that applying A' should produce the correct state. To prevent **H** from growing indefinitely, it is periodically cleared in periods of low-activity, which can be identified via distributed-consensus [8].

Algorithm

Algorithm 3.3 shows how OT is used in Scalabel to ensure that every client state converges to the server state once all actions have been received. It uses timestamps to find the correct place in history for the broadcasted action, then iteratively transforms the action against every action with a larger timestamp. To prove that this algorithm makes client state match server state, start with the following assumptions:

1. The transformation function works, so $T(A, B) * B = B * A$.
2. Every action in the history has a timestamp. This is already true for broadcasted actions, so it just requires waiting for the confirmations of pending local actions.

In the general case, there is some initial state S_0 and correct action history $H^* = [A_{t_1} \ldots A_{t_n}]$ where $t_1 \ldots t_n$ are the timestamps that define the ordering. The client receives some permutation $[A_1 \ldots A_n]$ of these actions via the SyncAction method. **Claim 1** is that after each call to this method, the following two conditions hold.

1. The local action history H is in temporal order, i.e. ordered by timestamps.
2. The local state is $S = H * S_0$, the result of applying every action in the local history to the initial state.

If this claim is true, the algorithm is correct, since after the last call to `SyncAction`, condition 1 says the local action history must be $H = [A_{t_1} \dots A_{t_n}] = H^*$. Then by condition 2, $S = H * S_0 = H^* * S_0$, which is the same as the server state.

Before proving **Claim 1**, we first prove the following lemma for the list inclusion transformation, which says that `LT` retains the correctness property of T.

$$\texttt{LT}(A, H) * H = H * A \qquad \textbf{(LT Lemma)}$$

By abstracting out the last transformation T against $H[n-1]$:

$$\begin{aligned}
\texttt{LT}(A, H) * H &= T(\texttt{LT}(A, H[0:n-1]), H[n-1]) * H[n-1] * H[0:n-1] \\
&= H[n-1] * \texttt{LT}(A, H[0:n-1]) * H[0:n-1] \\
&\vdots \\
&= H[n-1] * H[n-2] * \dots H[0] * A \\
&= H * A
\end{aligned}$$

The dots represent the repeated unraveling of the LT function in the same vein as the first step, which proves the lemma.

To prove **Claim 1**, use induction on the number of calls to `SyncAction`. After a single call, both conditions obviously hold. Now assume the claim holds after k calls, so the partial action history H is in temporal order, and $S = H * S_0$. Then the claim still holds after calling `SyncAction` for the $k+1$st time. First, H remains in temporal order because the new action is inserted into the list according to its timestamp. Next, the new state is:

$$\begin{aligned}
\texttt{LT}(A, H[m:n]) * S &= \texttt{LT}(A, H[m:n]) * H * S_0 \\
&= \texttt{LT}(A, H[m:n]) * H[m:n] * H[0:m] * S_0 \\
&= H[m:n] * A * H[0:m] * S_0
\end{aligned}$$

The last step follows from the **LT Lemma** and shows that the second condition holds since the new state is the result of applying the actions in their temporal order to S_0. This completes the proof that server and client states will eventually match, even if actions are delivered to the client in an arbitrary order.

3.4 Transformation Rules for Scalabel

Defining the transformation rules is a domain specific problem, unlike Algorithm 3.3 which could be adopted to any domain. In principle, the transformation function can be defined by n^2 rules for how every action transforms against every other, but in practice, this can be simplified by making actions commutative or parameterizing actions by some shared variable, so that transformation rules just describe how the shared variable changes.

Designing the specific rules for Scalabel requires a deeper understanding of the state structure and action effects. Figure 3.4 shows a minimal example of Scalabel state. This

```
{
    "ordering": ["ID-A", "ID-B"],
    "items" : [
        {
            labels: {
                "ID-A": BoxA,
                "ID-B": BoxB
            }
        }
    ]
}
```

Figure 3.4: JSON structure of a task with one item. Only a minimal subset of relevant properties are shown. Boxes are objects with string or numeric properties, like category name or position coordinates. The ordering list tracks which labels should be displayed in front/behind others, which is useful for marking occluded objects.

paper considers the subset of three task actions: addLabel, deleteLabel, and changeLabel. This seemingly small subset actually encompasses many labeling tasks, since adding a label has the same effect for 2D polygons as it is does for 3D bounding boxes. The effect of these actions on the state is:

1. To add a label to an item, a new ID is generated and added to the start of the ordering list. A key-value entry is also added to the appropriate labels dictionary. IDs are randomly generated strings with very low chance of conflict

2. To delete a label by ID, remove the ID from the ordering, and remove the key-value entry for that ID.

3. To change a label by ID, modify some properties of the box at the key corresponding to that ID.

Many pairs of actions are commutative, which simplifies the transformation rules. If two actions affect different labels, the only shared state they modify is the ordering list. ChangeLabel doesn't use the ordering list, and deleteLabel removes labels from the list by ID, so it isn't affected by changes to other labels. Thus, changeLabel and deleteLabel are commutative with each other and with themselves when the labels are different. Next, a label must always be added before it is changed or deleted, so addLabel is only affected by operations on different labels. Moreover, changeLabel on a different label will never affect addLabel.

The remaining transformation rules cover the non-commutative cases:

- Deleting the same label twice transforms the second deletion into a null action.
- Changing the same label twice is the case covered by the initial examples in Figure 3.1. The general solution is to override all properties shared between the two changeLabels. In this example, the incoming action A_1 is transformed so that it doesn't interfere with the effects of A_2, an already executed action with a newer timestamp:

$$\begin{aligned} A_1 &= \text{change}(\text{x} = 1, \text{ y} = 2) \\ A_2 &= \text{change}(\text{y} = 3) \\ A_1' &= T(A_1, A_2) \\ &= \text{change}(\text{x} = 1, \text{ y} = 3) \end{aligned}$$

- Deleting then changing the same label should prioritize deleteLabel. A common sequence of user actions might be to delete then redraw a label, so prioritizing changeLabel instead would result in duplicate labels. Transforming changeLabel into a null action achieves the correct effect:

$$\text{del} * \text{change} = \text{change} * \text{del} = \text{del}$$

- Adding two different labels is independent except for the ordering. The order of a label is used to decide whether it should be displayed in front of or behind other labels. This is useful for marking occlusion, and a similar property called z-index is used for layering in [22], an OT-based collaborative version of PowerPoint. New labels are inserted in the front of the ordering by default, which corresponds to an implicit parameter *orderIndex* with default value 0. The transformation rules describe how each action modifies *orderIndex*:
  - To transform against another addLabel, increment *orderIndex*, putting the label with the older timestamp further back in the list.
  - To transform against deleteLabel, decrementing *orderIndex* is incorrect because:

$$\texttt{LT}(\text{add}(\mathit{orderIndex} = 0), [\text{add(B)}, \text{delete(C)}]) = \text{add}(\mathit{orderIndex} = 1)$$

    since deleting label C should not affect the *orderIndex*. As a general rule, deletions cancel out additions, so it should only decrement on deleteLabel if the corresponding addLabel is also present.

3.5 Undo-Redo

Scope of Undo

Deciding which actions should be undo-able is more difficult than it first appears. The system has multiple points of entry, meaning users can edit different parts of the state

simultaneously. This makes global undo a bad choice, since users would accidentally undo each other's actions. One alternative is to localize undo to the current item, but instead for simplicity users are restricted to only undoing their own actions. Also, users are limited to only undoing their most recent action, unlike [21] which allows any action to be undone at any time.

Strategies

Undo-redo is a complex function, so there are several possible strategies. To illustrate how the final strategy was reached, some prototype strategies are also discussed.

History-based undo: The system tracks the state history, and undo/redo moves the state to different points along the history. With this strategy, the state history will grow very large, and reverting the state will undo everyone's actions, not just the local ones.

Undo via inverse actions: To undo, take the last local action from the history, and execute its inverse action. This requires actions to be invertible, which can be made possible by storing additional information with the action. For example, [21] saves the deleted characters in the history buffer in order to invert delete operations. In Scalabel, deleteLabel can save the label ID, so that undo can add a label with the same ID. However, redo will not work with strategy, since executing undo then redo will make the system re-execute the inverse operation.

Final strategy: The final strategy is shown in Algorithm 3.5. It is similar to the inverse action strategy, but it keeps the actions to be undone/redone in separate stacks from the normal actions. This approach nicely fits into the OT-based synchronization algorithm with no additional overhead. Clients only receive broadcasts of normal actions since the high-level undo procedures are only defined and interpreted locally on a per-client basis.

Fixing Transformation Rules

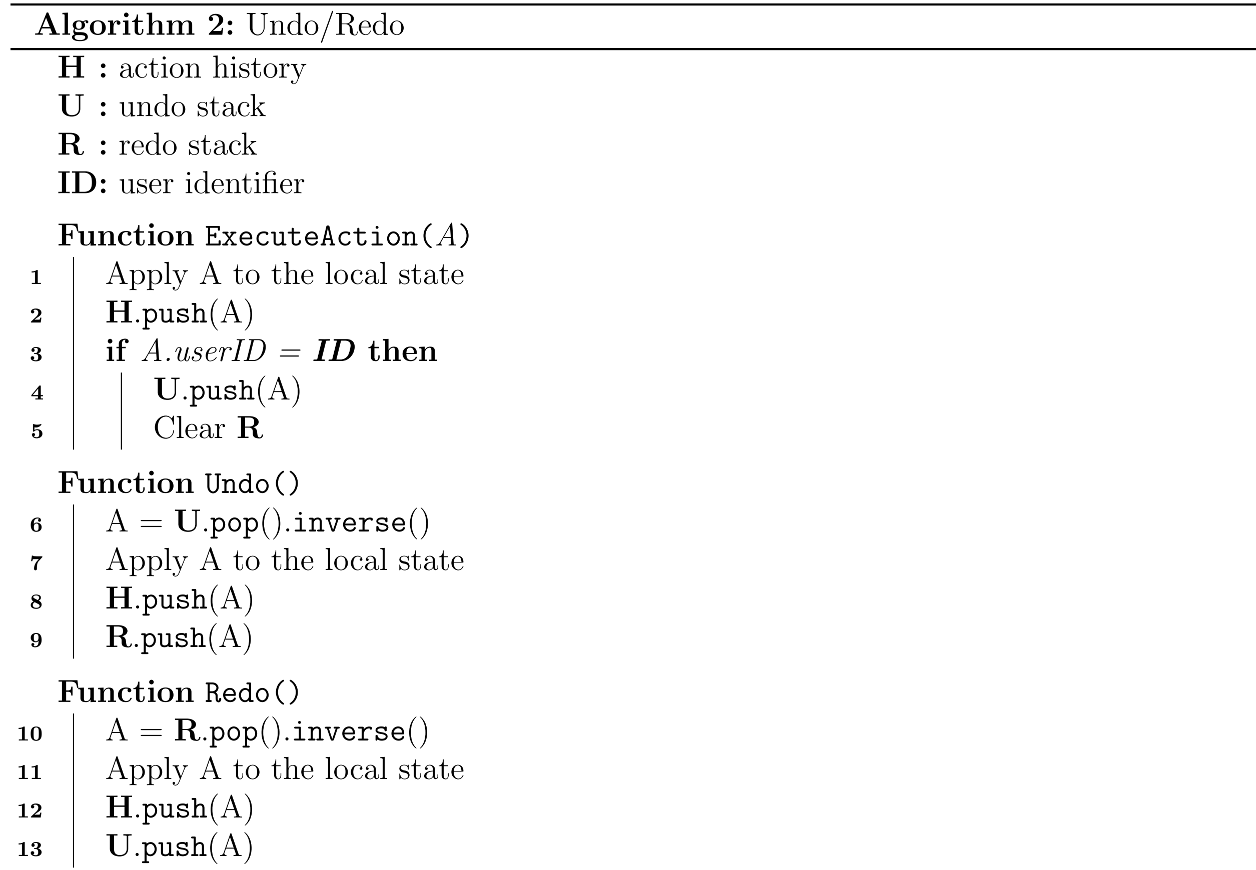
The final algorithm is designed so that OT can handle undo/redo conflicts without special cases, but there are still a few issues. The original transformation rules for Scalabel assume that deleting or changing a label before adding it is impossible, but with undo/redo, this becomes possible. For example, if user 1 takes the actions:

[addLabel(A), undo, redo]

This resolves to an action history of:

[addLabel(A), deleteLabel(A), addLabel(A)]

If user 2 tries to delete label A before the redo, user 1 needs to transform deleteLabel(A) against addLabel(A). For this situation, any effects before label creation should be nullified



Algorithm 2: Undo/Redo

```
H : action history
U : undo stack
R : redo stack
ID: user identifier

Function ExecuteAction(A)
1     Apply A to the local state
2     H.push(A)
3     if A.userID = ID then
4         U.push(A)
5         Clear R

Function Undo()
6     A = U.pop().inverse()
7     Apply A to the local state
8     H.push(A)
9     R.push(A)

Function Redo()
10    A = R.pop().inverse()
11    Apply A to the local state
12    H.push(A)
13    U.push(A)
```

Figure 3.5: The final undo/redo algorithm. Only actions that are originated by the local user are added to the undo stack. Undo and redo are high level functions locally, but ultimately execute normal actions which can be synchronized between users. The operational-transformation components are left out for simplicity.

by the transformation. This translates to the following requirements:

$$\begin{aligned} \text{T(del, add)} * \text{add} &= \text{add} * \text{del} \\ &= \text{add} \\ \text{T(change, add)} * \text{add} &= \text{add} * \text{change} \\ &= \text{add} \end{aligned}$$

These are satisfied by making both transforms produce null actions. To handle addLabel transforming against changeLabel and deleteLabel, the requirements are:

$$\text{T(add, del)} * \text{del} = \text{del} * \text{add}$$
$$\text{T(add, change)} * \text{change} = \text{change} * \text{add}$$

These are satisfied by setting T(add, del) to a null action, and T(add, change) to an action which adds the modified label. T(add, change) can't just be a null action because the label might not have existed before it was changed.

Undefined Behavior

Undefined behavior can arise when users collaborate too closely. For example, if user 1 adds a label, then user 2 deletes it, what should happen if user 1 presses undo? User 1 probably wants to undo the deletion, but since the system only considers local actions, it would undo the addition, deleting the label a second time. As discussed in [1], when user actions interfere, undo loses any sensible meaning, and the best option is to display an error message or suggestion on how to resolve the issue. For now, we acknowledge these cases as an inevitable result of conflicting user intentions.

3.6 Model Serving Example

Model serving is a form of human-machine collaboration in which model predictions assist and speed up human labeling efforts. The infrastructure described in Chapter 1 can serve predictions from any model once an interface between the system actions and the model input/output is defined. An example use case is instance segmentation, where each object in the scene is labeled with a separate polygon. Users label objects in the scene with bounding boxes, which provides a starting point for Polygon-RNN++ [2], an instance segmentation model. The model refines the bounding boxes to polygon predictions, which are returned to the web users as seen in Figure 3.6. With autosave, a polygon prediction request is dispatched for every new box, but with manual save, actions can be bundled together, allowing faster batch prediction.

3.7 Future Work

A number of improvements and expansions to Scalabel are left for future work. First, although the architecture for synchronization is completed, the implementation of the OT algorithm is a work-in-progress. Next, this paper only covers the transformations for label addition, change and deletion, so transformations for tracking operations remain to be defined.

Human-machine collaboration also has much room for expansion. The existing demonstration with Polygon-RNN++ only uses polygon prediction; adding the polygon adjustment feature would make the user experience even more interactive and efficient. Additionally, improving the API between system actions and model input/output will ease the process of adding more models for different tasks types, like 2D bounding box or tracking.

Lastly, many of the scalability measures discussed in Section 2.2 remain to be implemented, even though much of the infrastructure is in place. This includes running multiple

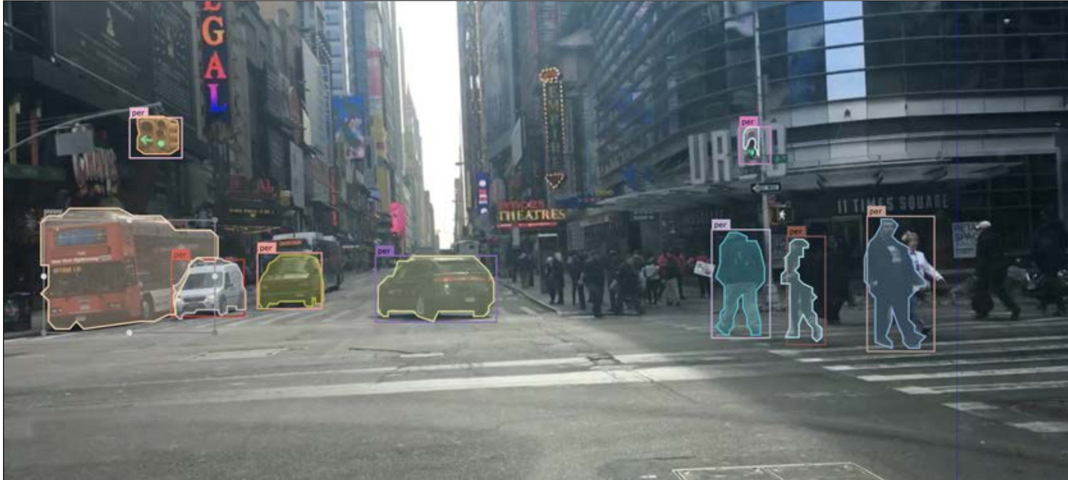


Figure 3.6: Polygon-RNN++ predictions synchronized to the Scalabel web user.

server processes in parallel, as well as automatically scaling and managing resource for machine learning models.

References

[1] G. D. Abowd and A. J. Dix. "Giving undo attention". In: *Interacting with Computers* 4.3 (Dec. 1992), pp. 317–342.

[2] D. Acuna, H. Ling, A. Kar, and S. Fidler. "Efficient Interactive Annotation of Segmentation Datasets with Polygon-RNN++". In: *CVPR* (2018).

[3] S. Amershi, M. Cakmak, W. B. Knox, and T. Kulesza. "Power to the People: The Role of Humans in Interactive Machine Learning". In: *AI Magazine* 35.4 (Dec. 2014), pp. 105–120. DOI: 10.1609/aimag.v35i4.2513. URL: https://aaai.org/ojs/index.php/aimagazine/article/view/2513.

[4] J. E. Archer Jr., R. Conway, and F. B. Schneider. "User Recovery and Reversal in Interactive Systems". In: *ACM Transactions on Programming Languages and Systems* 6.1 (Jan. 1984), pp. 1–19.

[5] S. Branson, C. Wah, F. Schroff, B. Babenko, P. Welinder, P. Perona, and S. Belongie. "Visual Recognition with Humans in the Loop". In: *European Conference on Computer Vision*. 2010, pp. 438–451.

[6] B. Collins, J. Deng, K. Li, and L. Fei-Fei. "Towards Scalable Dataset Construction: An Active Learning Approach". In: *European Conference on Computer Vision (ECCV)*. 2008, pp. 86–98.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[8] C. Ellis and S. Gibbs. "Concurrency Control in Groupware Systems". In: *ACM SIGMOID International Conference on Management of Data* (June 1989), pp. 399–407.

[9] J. A. Fails and D. R. Olsen. "Interactive Machine Learning". In: *Conference on Intelligent User Interfaces (IUI)*. Jan. 2003, pp. 39–45.

[10] J. Ferrié, N. Vidot, and M. Cart. "Concurrent Undo Operations in Collaborative Environments using Operational Transformation". In: *Cooperative Informations Systems (CoopIS)* (Oct. 2004), pp. 155–173.

[11] N. Fraser. "Differential Synchronization". In: *ACM Symposium on Document Engineering*. 2009, pp. 13–20.

[12] E. Kamar, S. Hacker, and E. Horvitz. "Combining human and machine intelligence in large-scale crowdsourcing". In: *Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. Vol. 1. June 2012, pp. 467–474.

[13] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565.

[14] T.-Y. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. "Microsoft COCO: Common Objects in Context." In: *European Conference on Computer Vision* (2014), pp. 740–755. Springer.

[15] T. Lindholm. "A three-way merge of XML documents". In: *ACM Symposium on Document Engineering*. Oct. 2004, pp. 1–10.

[16] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Joradn, and I. Stoica. "Ray: A Distributed Framework for Emerging AI Applications". In: *The 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Oct. 2018, pp. 561–577.

[17] G. Oster, P. Urso, P. Molli, and A. Imine. "Data Consistency for P2P Collaborative Editing". In: *Conference on Computer Supported Cooperative Work (CSCW)*. 2006, pp. 259–268.

[18] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors". In: *ACM Conference on Computer Supported Cooperative Work (CSCW)*. Nov. 1996, pp. 288–297.

[19] O. Russakovsky, L.-J. Li, and L. Fei-Fei. "Best of both worlds: Human-machine collaboration for object annotation". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 2121–2131.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *Conflict-free Replicated Data Types*. Tech. rep. 2011, p. 18. inria-00609399v1.

[21] C. Sun. "Undo as a concurrent inverse in group editors". In: *ACM Transactions on Computer-Human Interaction* 9.4 (Dec. 2002), pp. 309–361.

[22] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. "Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration". In: *ACM Transactions on Computer-Human Interaction* 13.4 (Dec. 2006), pp. 531–582.

[23] Valve. *Source Multiplayer Networking- Valve Developer Community*. 2019. URL: `https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking` (visited on 05/21/2020).

[24] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell. "BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[25] F. Yu, A. Seff, Y. Zhang, S. Song, T. Funkhouser, and J. Xiao. *LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop*. 2016. arXiv: 1506.03365.