Online Collaboration for Free: A Platform for Transparent Live Synchronization

Gilad Bracha
F5

November 17, 2025

Abstract

We describe Newspeak-on-Croquet, a programming system that enables the construction of collaborative applications without the need for the programmer to make them collaboration-aware. Clients written in the system automatically synchronize live, online. All synchronization between clients is handled by the platform. Because the platform's public API is identical to its non-synchronizing predecessor, pre-existing applications designed and built without any collaboration support can be run on the new platform unchanged, and automatically become meaningfully collaborative. We demonstrate the efficacy of our design by taking a substantial existing application, the Newspeak IDE [Tea21], and running it on the new platform as a collaborative IDE.

1 Introduction

Modern personal productivity applications (e.g., Google Docs, Microsoft Office) often support online collaboration between multiple users. These applications synchronize distributed instances of themselves so that users in different locations all see the same application state as they interact with the system. Such applications must typically be programmed to support this synchronization explicitly. The design and implementation of such support is non-trivial. Various libraries are used to simplify this task. However, programmers must still manually interact with these libraries and design and code accordingly, posing a substantial challenge. The Newspeak-Croquet platform relieves this burden by allowing any interactive application to become collaborative, even if the application was not conceived or coded as such.

This paper does not deal with offline synchronization. Therefore, it does not address the problem of producing local-first [KWvHM19] applications, where clients may continue to do work while offline, and the resulting divergent replicas are reconciled when they come back online.

Our implementation uses the Newspeak programming language [BvdAB+10] and the Croquet synchronization service [OLE+22], but the basic ideas should carry through to other systems. The core idea is to engineer online synchronization into the internals of the UI framework, without exposing synchronization in the API. As a result, programmers code a traditional GUI application, completely oblivious to synchronization and/or collaboration.

In the next section, we provide useful background about both Croquet [§2.1] and Newspeak [§2.2]. This is followed by detailed description of our approach [§3]. Then we discuss possible objections and concerns that might be raised and our response to them [§4]. Related [§6] and future work [§7] and conclusions [§8] come at the end.

2 Background

2.1 Croquet

Croquet is a programming system that facilitates the creation of online collaborative applications. The current version of Croquet is implemented in Javascript and runs on the web. This technique is reminiscent of work in gaming. It has its root in earlier work in Smalltalk [GR83]. Our interest in Croquet is in its synchronization support.

Croquet allows clients to reliably and scalably synchronize when online. Each client responds to input by publishing events to a server, often referred to as *reflector*. The server orders incoming events based on a notion of virtual time that it maintains, and then sends the event data to all clients, ordered by virtual time. Clients then act on the events. All clients start in the same state, and the platform ensures that they respond to all events deterministically. This ensures that all clients execute in lock-step, maintaining the same underlying state.

Clients are free to display the state in any way they choose. It is the underlying state of the application, known as the *model* which is synchronized. Note that Croquet does not support offline work. Clients may go offline, but they are then frozen until they come back online, at which time they can receive an act on any events that occcurred since they disconnected.

Using only event reflectors, long lived sessions would be very costly to join. Imagine joining a chat room that has been running for years. One would have to play all the evnts that occurred over that period, resulting in painful latency. To address this issue, Croquet periodically takes a snapshot of the model and stores it on the server. When a client subsequently joins, it is restored from the snapshot, and only subsequent events are reflected to it so it can catch up.

2.2 Newspeak

Here we review the characteristics of Newspeak that are salient to this work. Newspeak is a class-based language in the Smalltalk tradition. It has a Smalltalk-like syntax that relies heavily on keyword-based notation.

Newspeak supports class nesting and uses classes as its main structuring construct. Where most languages would use a module/package/unit construct, Newspeak uses top-level classes. A relevant example is Hopscotch, Newspeak's UI library, which is realized as a class in which various UI classes (e.g., classes for buttons, windows, menus etc.) are nested.

All names in Newspeak are late-bound. In particular, class names are late bound, enabling classes to be overridden like methods. Thus Newspeak supports class hierarchy inheritance [OH92, Ern03].

Newspeak is an object-capability language with no global namespace. Top-level classes can only access the outside world outside via arguments passed in to their factory methods, which are somewhat analogous to constructors. Notably, standard utilities like collections, files, the UI etc. must be provided externally. This is commonly done by passing in a platform object that aggregates these capabilities. Different implementations of these core libraries and utilities can be used, by passing in different platform objects.

The standard way to define platform objects is via *runtime classes*. Examples would be classes such as RuntimeTimeForPrimordialSoup or RuntimeForHopscotchForHTML. The former provides a bare bones platform without a GUI or reflection. The latter is the standard one used in interactive applications; it supports a version of the Hopscotch GUI that runs on top of the DOM, as well as reflection, graphics etc. There are others.

Newspeak applications are objects that support a standard interface, comprised of the method main: platform args: args which accepts a platform object and a collection of additional arguments (analogous to argv in C). The role of this method is to initialize and run the application. The process typically involves instantiating the applications constituent class libraries, often passing the incoming platform argument down to them.

Applications are defined via classes that also obey a specific interface: the class object must have a factory method named packageUsing: that takes in a *manifest* object. The manifest provides access to any libraries that the application requires that are not provided of the platform object. Deploying an application involves instantiating the application class and serializing the resulting object.

The current implementation of Newspeak runs in the web browser. The usual way to run a Newspeak application today is to access a URI which runs Newspeak, and specifes both the desired platform and application as a single binary file that combines the two. Newspeak provides tooling that allows one to produce such a binary from the desired platform, application class and manifest.

2.2.1 An Example Application

To illustrate the use of the platform for collaboration, we present a minimal example, a collaborative counter, trivially derived from a non-collaborative one. The counter presents the UI shown in figure 1, consisting of the counter value on the left, and buttons for incrementing, decrementing and resetting the counter to zero.

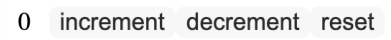

Figure 1: Counter UI

The code that generates the UI is shown below:

```
row: {
    label: subject count.
    mediumBlank.
    button: 'increment' action: [updateGUI: [subject increment]].
    button: 'decrement' action: [updateGUI: [subject decrement]].
    button: 'reset' action: [updateGUI: [subject clear]].
    }.
```

The full code for the counter is in a class called CounterUI available at https://github.com/newspeaklanguage/newspeak/blob/master/CounterUI.ns.

To deploy the counter as a newspeak application, we package it as an application in the following class [1]:

```
class CounterApp packageUsing: manifest = (
|
   private CounterUI = manifest CounterUI.
|
) (
public main: platform args: args = (
    |
    ui = CounterUI usingPlatform: platform.
    |
    platform hopscotch HopscotchWindow openSubject: (ui CounterSubject on-
Model: ui Counter new).
)
)
```

[1] Again, this code is publicly available at https://github.com/newspeaklanguage/newspeak/blob/master/CounterApp.ns.

Note that nothing in the code has anything to do with synchronization or collaboration. Indeed, this example long predates the work on synchronization described in this paper.

The only difference between a collaborative application and a non-collaborative one is the choice of platform made in the deployment step. Usually, this code is deployed using a platform object provided by RuntimeForHopscotchForHTML. If we take this exact same code and deploy it with the RuntimeForCroquet platform object, it will synchronize over the network so that each change made to the counter in any client is reflected live across all clients. In the next section, we will explain how RuntimeForCroquet's platform differs from the standard one.

3 Integrating Croquet into Newspeak

There are different ways one might integrate Newspeak and Croquet. The most obvious is simply to allow Croquet to be used from Newspeak in the way it is usually used in Javascript. This is not our intent.

The normal use of Croquet assumes a *collaboration-aware* application, which can be associated with various UIs in different clients. In such an application, Croquet is used at the application's semantic level (i.e., as part of the *model*). Our goal is to enable collaboration with *collaboration-oblivious* applications. The trade-off is that we require all clients of the application to have a common UI. Croquet is then integrated into application at the level of the *view,* by using Croquet as part of the UI library [2]

To achieve the above, we provide an alternate implementation of Hopscotch, the standard Newspeak user interface library. This variant implementation has the same API as the original, but differs in that every interactive UI element works through Croquet. Consequently, a series of clients running a Newspeak application can be synchronized, without any modification to the Newspeak application code; the only requirement is that application be deployed using a platform object that uses the revised UI implementation.

The revised platform is implemented by RuntimeForCroquet. Its main difference from the standard platform is that it uses HopscotchForCroquet instead of the standard HopscotchForHTML5. How does HopscotchForCroquet work?

Every kind of UI widget in Hopscotch is implemented by a class. Our main concern is with those widgets that directly implement user interaction. Examples would be buttons and hyperlinks of various kinds, checkboxes, menus, sliders etc. Each of these widgets must notify Croquet of an event signifying its use and wait for Croquet to signal back; only then should it perform its normal function.

Concretely, each widget class subclasses the corresponding class in HopscotchForHTML5 overriding the interactive behavior of the superclass. Rather than performing the normal action, it publishes a Croquet event with suitable data. When Croquet responds, the widget will invoke the overridden code in

[2] We are keenly aware that his runs somewhat contrary to the intent of the Croquet framework. See section 4 for more details.

the superclass. The scheme is designed so that we only generate events upon user interaction. Thus the event pressure is bounded by the speed of human input.

Fundamentally, our approach is not dependent on using Newspeak or Hopscotch. One could achieve the same effect by revising a UI library in other languages; [3]it would just be a lot more difficult. We discuss the advantages of using Newspeak in section 4.3.

There are a number of subtleties that complicate the simple scenario we have just described. We discuss these in the following subsections.

3.1 Snapshotting

Croquet's use of snapshotting poses a difficulty for us. All our state is in the Newspeak heap, not in the Croquet model. A snapshot of the Croquet model therefore contains no useful information that would enable client restoration. In addition, the Croquet snapshotting process is restrictive, as it requires all data to be serializable according to the rules of Javascript. It cannot naturaly deal with Newspeak state such as closures, for example.

Currently, we address this by explicitly storing all events in the Croquet model ourselves. When a client joins, we replay any events that occurred since the client was last active. This does suffer from the very weakness that Croquet's snapshotting model was designed to avoid, but enables clients to (re)join a session after it has already started.

An alternative would be to snapshot the Newspeak heap (much like a Smalltalk image) and store that as a byte array/blob in the Croquet model object. The current Newspeak implementation does not include a Smalltalk-style image however. Moreover, Croquet snapshots rather frequently and offers no way to control this frequency. Under such circumstances, the cost of producing an image and sending it to the Croquet server might prove prohibitive.

3.2 External System State

Our approach is predicated on all clients always being in the same state. This invariant may be undermined when clients access their local system, whose state may vary and is beyond our control.

3.2.1 File Access

One example is interaction with the file system. Consider what happens when a user asks to load a file (e.g., reading a source file into the IDE). If all connected instances get the request, they will each open a file dialog on their local file system, which may or may not have the same (or any) version of the desired file, in the same (or any) place. In practice, the browser won't even allow the file dialog to even open except on the initiating client.

[3]Some additional adjustments are needed to deal with interactions with local state such as the file system. See section 3.2.

A solution is to encapsulate file access in a Newspeak API, that is overridden for Croquet. The Croquet version opens a file selection dialog as usual on the initiator. Then, rather than commence the usual processing of the file(s) it has read, it saves the loaded file data on the reflector, and then posts an event indicating that file loading. Other clients do nothing vis-a-vis the OS when the file API is called, but are set up to listen for the data and act on it when it arrives.

This worked, but we faced size limits on the amount of data a Croquet event will accept, so it only works for small files. To fix this, we use another Croquet API, that lets us store data on the server under a handle. We pass the handle via an event, and then each client can retrieve it.

Similar handling is necessary for features such as drag and drop of images or other file types.

3.2.2 Web Browser State

Browser state differs between clients at application start. A prime example of such state is browser local storage.

As an example, the Newspeak IDE uses local storage when saving, and to track changes to the codebase. Upon startup, it checks the storage. If there are any stored saves or changes, it brings up a screen asking the user if they would prefer to start from the latest backup, the latest save, or the default clean state. If there are no saves or changes, it skips this screen and starts up the in the clean state directly.

The problem is that a new client will have no saved state, whereas a client rejoining a session might. This difference in state is reflected in the UI, and undermines the assumption that all clients execute in lock step.

In general, a collaborative application that relies on web browser/JS state directly cannot function reliably. Hence, we need to have the Newspeak platform mediate access to local storage. This shows that it is not always sufficient to control synchronization via the UI. Under this regime, the IDE queries the platform to see if there is any relevant data in local storage. The platform sees to it that local storage is empty at application startup (virtual time 0). After that, local storage is modified by application logic in lock step.

Another form of browser state is the history of the tab. The browser back/forward buttons are used to enable convenient navigation within the application, and so they impact the Newspeak UI being displayed. Therefore they must be sync'ed. If we step back to an an earlier page, unrelated to the application, or if we have links within the application that take us elsewhere the application behaves as if its tab were closed or hidden. Reopening or returning to the tab will bring it up in the correct state (the latest state of any active client).

3.3 Localizing Views

There are cases where it is not necessarily desirable to keep the UIs precisely synchronized across clients. Individual users are likely to be using different devices with different screen sizes. They may have different vision needs. We therefore avoid synchronizing scroll bars. Likewise, controls over zooming in and font size are not sync'ed. It might be best to allow user control over these behaviors, in cases where synchronization of these elements is desired.

Croquet's original model, where only the model is synchronized, gives more leeway in this regard. One can bind completely different UIs to a shared, synchronized model. However, any state that one wishes to synchronize needs to be preserved in the model. If one wanted to synchronize scrolling or zooming, one could not completely offload this behavior to lower levels of the UI stack. This is a general problem with user interface libraries that maintain all state in the model (like Elm [CC13]).

Being able to decouple views, at least partially, has important advantages for privacy and security. We allow views to be partially decoupled, so that users can interact with private data. Our approach to these issues is discussed in section 4.2.

4 Discussion

4.1 Collaboration Aware vs. Collaboration Oblivious

Our approach is not entirely aligned with Croquet's design philosophy. Croquet assumes that an application is collaborative by design. The idea of collaboration-oblivious applications being collaborative is frowned upon.

An important argument in favor of collaboration-aware applications is that, given two events, A and B, such that event A makes event B semantically incoherent, if different clients initiate events A and B in close succession, then an invalid situation may occur and the application must be aware of the possibility to defend against it. As an example, consider the following scenario:

User 1 deletes class C (event A). User 2 adds a method to class C (event B) at exactly the same time (at some low-millisecond resolution of time). Now if B is handled first in virtual time, we get a method added and then the whole class deleted. However, if A precedes B in virtual time, we end up attempting to add a method to a class that has gone missing, possibly crashing the application.

The counter-argument is that people are collaborating in real time. They are mutually aware that they are sharing a resource. Just as, in a physical meeting, one does not start erasing the whiteboard while a colleague is drawing upon it, one should not engage in contentious activities during live online collaboration.

However, we must allow for accidental contention (say, someone clicks a button by accident). A solid undo facility may be helpful. In addition, we may guard against such accidents in other ways. These accidents are essentially race conditions, and we can deal with them using the analog of a global interpreter lock. We can realize this by maintaining a generation counter per client. Events

come with their generation number, and if an event comes from a generation already seen, it gets dropped, preventing such race conditions from happening.

Another objection is that an application that is not collaboration-aware will not incorporate features specifically designed to leverage collaboration. For example, they won't show distinct user cursors in an editor. As another example, there are less likely to incorporate support for, say, commenting on or reviewing a text.

There are two responses to this objection. First, common collaboration patterns like multiple user cursors should be provided by the underlying UI framework, so the application need not take special action to implement them. Second, there is nothing preventing an application from adding specific collaboration-aware features. It is in fact much easier to do that starting from a collaborative baseline provided by the platform.

Security and privacy are additional topics that may require specialized attention that a collaboration-oblivious application will not provide, as discussed below.

4.2 Security and Privacy

Under the Croquet model, all clients are in the same state. This makes it harder to provide different users with different privileges, as all data is available to everyone. User specific data can be kept outside the shared model however. This would allow for having a notion of a *current user* and using the UI to control data access on that basis. Such a notion only applies in a collaboration-aware setting, but is clearly useful.

However, in our system the UI itself is executed identically on all clients. How are we to arrange for distinct display of information to individual users?

To address this, we take advantage of the fact that the non-synchronizing UI still underlies the synchronizing one. We provide a non-synchronizing view class, that provides access to the original, non-overridden, non-synchronizing widgets defined by HopscotchForHTML5. This view class, NonSyncingPresenter, may be subclassed to define UI elements that do not synchronize. However, such UI components must not be used to modify any state that is shared between users. Otherwise, the synchronized UI would no longer be running on the same shared state, with unpredictable and inconsistent results.

The above approach supports private, per user UI elements. What happens when we want to collaborate with distinct subsets of users, keeping the interaction hidden from some but not others? One can extend the idea of specialized views to which either target distinct Croquet sessions, or leverage Croquet's ability to target events to specific "scopes". [4] We leave it for future work, discussed in section 7.

[4] In the extreme case, private views are ones whose scope only inclues the current client, but in that case, we need not send events to Croquet at all.

4.3 Advantages of Using Newspeak

As noted in section 3, our work does not fundamentally depend on the choice of language or UI framework. Nevertheless, the use of Newspeak has facilitated our project in two interesting ways.

First, class hierarchy inheritance made it much easier to adapt an existing UI library to support synchronization internally without changing the external interface. Using a conventional language, it would considerably more difficult to subclass the various classes in the library as we have done here.

Clients would normally be dependent on the class names of the original library, using them as constructors and types. Some might overcome these dependencies by using dependency injection frameworks, but this could not be relied upon. Thus subclasses would have to have the same names as the originals (their superclasses). This isn't actually possible - the subclasses would have to differ in their package names.

A forked library might choose to rename all the original classes, and then subclass them under the original names, preserving the API for applications. It might be less error prone to rename the original UI package(s) (so as to minimize the number of internal dependencies that need renaming) and use the old package name for the new library. Alternatives, such as modifying the original classes in situ, are far worse, as internal dependencies such as overridden method and class names need to be consistently renamed in a rather error-prone refactoring.

Second, the use of platform objects means that we do not have to recompile or relink applications as would be necessary using languages such as Java, C#, Go, Swift etc. The flexibility provided by the platform object is a direct consequence of using a strict object-capability language.

Lastly, in many languages, some platform functionality is not always readily replaceable. On the web, file system interaction is mediated by the UI, but in other systems this is usually not the case, and in some languages it may be harder to disentangle it from the rest of the system.

5 Status

Newspeak-Croquet is a working prototype. We have used it successfully to run a variety of applications, all of which predate the Croquet integration and are thus necessarily collaboration-oblivious. These applications include simple demonstrations like the interactive counter described in section 2.2.1, or TodoMVC, but extend to sophisticated systems like the complete Newspeak IDE and the Ampleforth [Bra22] live document system. As such, it supports applications built with these systems, such as the Telescreen presentation manager or the Room101 blog [Brab].[5]

Not all the ideas described in this paper are fully implemented at this time. In particular, the global lock to protect against accidental conflicts, user cursor

[5] For a demonstration of a somewhat earlier version, see [Bra25].

tracking and multiple cursors are not yet working.

6 Related Work

The notion of relieving applications of the burden of synchronization is a form of *orthogonal synchronization* [Braa]. However, the original vision for orthogonal synchronization sought to use the programming language as its basis. Here we rely on the platform rather than the language. Furthermore, orthogonal synchronization was supposed to cover offline work, which we do not cover. Lastly, this work is distinguished by a actually having a fully working implementation.

Orthogonal synchronization was inspired by orthogonal persistence [AB87], which remains quite relevant with respect to efforts to extend this research to offline work. The topic of synchronization of offline clients is the focus of local-first applications [KWvHM19]. The bulk of that work relies on the use of CRDTs [SPBZ11] and includes various frameworks to support such synchronization [Kle, NJDK16, YJS].

Another approach to synchronization is the use of operational transforms [], popularized by tools such as Google Docs [].

Webstrates [KEB+15, BFG+22] uses its own approach, based on synchronizing the DOM.

Source control systems also involve synchronizing distribute replicas and the ideas have been applied to more general applications [KSJ19].

A common approach to collaboration tools, especially those aimed at software developers [Tup, CoS, RSFWH98, OFA17] is based on sharing a display across clients. Superficially, there is a similarity to our approach, in that input events are captured across clients. However, only one client is actually running the underlying application. The others see that client's display, which is demanding on network bandwidth. No state is shared across clients. If the shared copy becomes inaccessible, there is no way to recover it. A consequence of this dependence on the network is that there is no path to local-first software using these tools. The advantage is that any application can be shared in this manner. However, each client must install the collaboration software.

7 Future Work

The most important area we wish to pursue is extending this work to support offline work. Of course, this greatly complicates the synchronization problem. Whereas we currently guarantee that replicas never diverge, once offline work is allowed, that guarantee is lost, and one has to be able to reconcile the divergent replicas. Whether we can achieve such synchronization without burdening the application programmer remains to proven.

Offline synchronization will likely require rather different techniques for synchronization. Our hope is that we can handle common situations, where the replica's are not in conflict, automatically, and ask the human collaborators to

resolve their conflicts, providing them with a collaborative diff view to assist them. It may also be feasible to push much of that burden onto AI.

Is there any place for our current approach once that alternate synchronization technology is in place? At this time, the answer is unclear. It may be that the new approach will subsume online synchronization. Conversely, it might be the case that our current approach works better when online, and it is best to keep two distinct subsystems for online and offline synchronization.

As discussed in section 4.2 there is much work to be done to support notions of privacy and security between clients. Newspeak is an object-capability language [Mil06], and we would expect to leverage that property extensively when working on security.

8 Conclusions

We have demonstrated the feasibility of a platform that supports live online collaboration in a manner that is transparent to applications. Our working prototype, Newspeak-Croquet, allows for collaboration-oblivious applications to support a useful level of online collaboration, without precluding collaboration-aware extensions. We have taken pre-existing, non-collaborative applications and made them collaborative without modification of the application source. Our examples range from trivial applications like counters or TodoMVC, to substantial ones like the Newspeak IDE, the Telescreen presentation tool and the Ampleforth live document system.

The strategy we have used is focused on modifying the user-interface library so that it encapsulates a live synchronization protocol that is not exposed to the application. Additional elements of the platform must also support encapsulated synchronization to prevent exposure of external local state to clients. We believe this strategy should carry over to other programming languages and their UI and other core libraries, albeit with greater difficulty due to their weaker modularity support.

Acknowedgements

I am especially indebted to the late Vanessa Freudenburg for her work developing the underlying Croquet system on the web. In addition Vanessa, as well as Geoffrey Litt, Yoshiki Ohshima, Peter van Hardenburg and Alex Warth provided many helpful comments on this work.

References

[AB87] Malcolm Atkinson and Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, pages 105–190, June 1987.

[BFG+22] Marcel Borowski, Bjarke V. Fog, Carla F. Griggio, James R. Eagan, and Clemens N. Klokmose. Between principle and pragmatism: Reflections on prototyping computational media with webstrates. *ACM Transactions Computer-Human Interaction*, 2022. Just Accepted.

[Braa] Gilad Bracha. Objects as software services. Invited talk at OOPSLA 2005 Dynamic Languages Symposium; updated video available at https://www.youtube.com/watch?v=_cBGtvjaLM0 Unpublished manuscript available at http://bracha.org/objectsAsSoftwareServices.pdf.

[Brab] Gilad Bracha. Room 101. Blog at https://blog.bracha.org.

[Bra22] Gilad Bracha. Ampleforth: A live literate editor. In *Live 22: The Eighth Workshop on Live Programming*, 2022. Available at https://blog.bracha.org/AmpleforthLive22/out/primordialsoup.html?snapshot=Live22Submission.vfuel. Recorded talk at https://www.youtube.com/watch?v=gfbzC_90fJE.

[Bra25] Gilad Bracha. Online collaboration for free. Video, August 2025. Presentation to the UK Smalltalk Users Group. Available at https://www.youtube.com/watch?v=hHvIEFy1v5A.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *European Conference on Object-Oriented Programming*, June 2010.

[CC13] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. *SIGPLAN Not.*, 48(6):411422, June 2013.

[CoS] Coscreen. Available at https://www.coscreen.co/.

[Ern03] Erik Ernst. Higher-order hierarchies. In Luca Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[KEB+15] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, 2015.

[Kle] Martin Kleppmann. Automerge. https://automerge.org/.

[KSJ19] Gowtham Kaki, KC Sivaramakrishnan, and Suresh Jagannathan. Version Control Is for Your Data Too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[KWvHM19] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 154178, New York, NY, USA, 2019. Association for Computing Machinery.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[NJDK16] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. pages 39–49, 11 2016.

[OFA17] Yoshiki Ohshima, Bert Freudenberg, and Dan Amelang. Kanto: a multi-participant screen-sharing system for etoys, snap!, and gp. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience*, PX/17.2, page 710, New York, NY, USA, 2017. Association for Computing Machinery.

[OH92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 25–40, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[OLE+22] Yoshiki Ohshima, Aran Lunzer, Jenn Evans, Vanessa Freudenberg, Brian Upton, and David A. Smith. An experiment in live collaborative programming on the croquet shared experience platform. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*, Programming '22, page 4653, New York, NY, USA, 2022. Association for Computing Machinery.

[RSFWH98] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the*

13th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS'11, page 386400, Berlin, Heidelberg, 2011. Springer-Verlag.

[Tea21] Newspeak Team. Newspeak IDE on Wasm, 2021. Available at https://newspeaklanguage.org/samples/primordialsoup.html?snapshot=HopscotchWebIDE.vfuel.

[Tup] Tuple. Available at https://tuple.app/.

[YJS] Yjs. Available at https://yjs.dev/.