# Shopping Lists on The Cloud

Large Scale Distributed Systems

Ricardo Inácio[1] Diogo Lemos[2] Tomás Maciel[3]

FEUP
{up202302742[1], up202003484[2] up202006845[3]}@up.pt

**Abstract**

Distributed solutions over the internet can be very complex, due to the sheer amount of mechanisms they should implement, including the support of concurrent control of data, replication, and fault tolerance. In this paper, the group presents a simplistic approach for a large-scale distributed shopping list. The system aims for a local-first[5] implementation, where data should always be accessible and persistent on the client's machine, and the application should run independent of connectivity status. It allows for concurrent read/write operations, with scalability always in consideration, in order to avoid data access bottlenecks. All operations are done on JSON objects, that constitute the shopping lists in question, identifiable by unique ID's (uuid4). They are shared across a network of clients and servers, that must always reach a middleware proxy, while using the **ZeroMQ** messaging library [2] for communication. To handle versioning, the service uses an implementation of a **Last-Writer-Wins CRDT** [4], which supports **add**, **remove**, **compare** and **lookup** functions, as well as **merging**, needed to resolve conflicts. The distributions are managed by a **consistent hash ring**, where all available nodes are uniformly mapped in a topological ring, being each responsible for storing and replicating, a list of data (keys),
**Keywords:** collaborative, distributed, replication, concurrency, CRDT, LWW, local-first, hash ring.

## 1 Communication Process

To be able to label this application as collaborative, it must be capable to reach other entities in the network, to transmit a client's lists, as its respective modifications, and to receive new ones, or simply the changes to local outdated versions. All this is achievable by defining a **communication process** that is built on top of a specific **protocol**, that must be followed by all participants.

In the presented service, there are three different entities that participate in the process: the *servers*, which store their designated data (keys) elements, the *clients*, which are able to *create* and *modify* (by adding, removing and checking elements on) shopping lists, and the *proxy*, which acts as a coordinator, by storing the connected servers, handling uniform data distribution by using a **consistent hash ring**, and making sure the correct entities receive the desired data.

The process starts when the ***service.py*** script is executed, and creates an instance of the **Proxy** class, as well as five independent instances of the **Server** class. Each of the latter is programmed to automatically send a "*S_HELLO*" message to the proxy's *socket_hello* (zmq.DEALER) socket until it's assigned to a hash ring position, which is confirmed with an "ACK" message on the *socket_ack* (zmq.SUB) socket by the **Proxy** instance. Then, each server starts listening on the *socket_r* (zmq.SUB) socket for incoming lists, so that it can perform its role-specific operations. At the same time, the server polls this socket alongside the *socket_reach* (zmq.SUB) for **S_REACH** messages, so that it can confirm to the proxy that it is active, and able to communicate, replying with an **S_ONLINE** message through the *socket_online* (zmq.DEALER). It should be noted that each of the *SUB* sockets are given as an option the server's ID, so that only messages specifically meant to arrive to the instance are delivered. The choice to create different sockets for the hello/ack and constant listening processes stemmed from the occurrences where a server started receiving messages/lists from clients, before being assigned a position in the hash ring, since it hadn't received its *"ACK"* message, blocking the service indefinitely.

Afterwards, the service is ready, and able to receive requests, messages, and lists from **Clients**, while also being always listening for new servers that may eventually join the ring, and verifying if the current active ones didn't somehow go offline, and need to be removed from it, all while managing reallocation process, so that data remains consistent and available.

To assure connectivity, each request is tried three times, because the system is designed so that if a message is not delivered (and a response is not received), it should be forwarded to the next candidate, and since every data point is replicated three times, the group thinks this is enough to assure at least one valid reply

The process, in which lists are shared, is based on a concept named **serialization**. In *ZeroMQ* (and in most messaging frameworks) the data sent amongst communicating entities needs to be encoded in a specific format, normally **bytes**. While testing, text strings were converted simply by calling the **encode()** function, and reversed using the **decode()** one. With this in mind, the group thought about using serialization, paired with functionalities provided by classes in OOP (Object-Oriented Programming), since every entity in the project, (and most importantly, lists) are represented as classes. By using the **pickle** python module, it's possible to convert an instance state into a *stream of bytes*, which can then be sent by *ZeroMQ*, and the receiver, using the same module, can deserialize the list's state into another (unrelated) instance, where every attribute maintains the same value. This approach was preferred to merely sending the JSON object, because then both sender and receiver can immediately start manipulating and using the list, taking advantage of the methods provided by the respective class, including saving to a local file.

Although this work is based on an implementation of a distributed concurrent system, in order to be able to develop it and test it, the setup of a simulation of a multi-user environment was needed to progress. To solve this issue, the group opted to simulate independent machines (specifically their local storages), by naming folders based on the entities ID (e.g. if client 1 sends a list that must be stored on server 1, then it must be created on the directory *client_1* and saved on *server_1*). All lists and "machine" directories can be found on the ***storage*** folder on the repository / project.

## 1.1 Servers

In this service, **servers** take the role of storing the latest version of lists, which have the server's hashed ID present in their respective **preference list** (list containing the node positions in the ring that are responsible for storing a range of keys). When *hashing* the list's key, it's possible to determine which are the positions on the ring where the servers that should store said list are located, and since these positions are evenly distributed, the system makes sure no section gets assigned more keys, and subsequently lists. It is also very efficient to perform this *lookup* operation, by using **Binary Search** (*O(log N)* where $N$ is the number of nodes), since the ring represents a sorted sequence.

As stated before, after the hello/ack phase of the process, the **server** instance is ready to start listening for lists, which it does by connecting to a subscriber socket *socket_r*. When a list arrives, if a previous version is already present on storage, it immediately performs the *merge* operation supplied by the **ShoppingList** class (which will be explained further in this paper), where basically it verifies the CRDT's *add* and *remove* sets, to determine which elements should be present in the final list.

When requested, and online, a server will send the latest version of a list (noting that this request can be intentional, as a *get* operation by the client, or simply an automatic *fetch*), by using its *socket_s* dealer socket, which is connected to the **proxy**, that should handle the routing from there.

As it is obvious, a server can only receive and reply to messages if it's online. In order to inform the **proxy** that fact, two sockets were implemented, *socket_reach* (zmq.SUB), and *socket_online* (zmq.DEALER), where the first is always listening for **S_REACH** messages from the proxy, and the latter ready to send **S_ONLINE** messages to ensure its presence in the network, and availability to handle requests.

To avoid bottlenecks and blocking, a **poller** was implemented on both subscriber sockets present, which handles correct delivery of messages to the respective destination. This mechanism will be further described in the Proxy section 1.3, since its implementation is more relevant to the project.

## 1.2 Clients

The **client** role in the network represents the users of the application, whom perform various activities by taking advantage of the functionalities provided by the class of the same name. Although the final user interacts with the **User Interface**, detailed in its own section 5, the **Client** class is the one doing most of the "behind the scenes" forwarding work. It possesses two different sockets: *socket_s* (zmq.DEALER), to execute requests (in a non-blocking manner), and a *socket_r* (zmq.SUB), to receive the latest versions of the lists. For the behaviour of the latter, two scenarios were considered: subscribe to messages that only contain its own ID (uuid), meaning it was a direct request, and subscribe to messages containing lists

ID's present on the local machine, to keep receiving updates to every single one. Various experiments were done to be able to test and possibly implement these ideas.

The main function present in this class is the **send_data()**, which is self-explanatory. It sends data (normally a list) and waits for a *SAVED* message. When needed, if the user knows the identifier (uuid) of the list, it can directly request it to the proxy, so that it forwards to the respective node, and retrieves it, being able then to save it on local storage.

## 1.3 Proxy

As the intermediary of this process, the **Proxy** must perform all the routing between all participant entities in the network, while being specially keen to the behaviour of **servers**, as they join and leave (intentionally or not) the network, since it is accountable for their correct allocation and positioning in the *hash ring*.

It possesses **eight** different sockets, two for the **clients** (a *frontend_r* (zmq.ROUTER) and *forntend_s* (zmq.PUB)), simply handling the arrival, processing, and correct routing of lists and messages from and to (respectively) the correct instances. The use of a *publisher* socket stems from the idea that a client must receive that latest version of list it requested, so the list's key is used as a parameter for the client's *subscriber* socket as well. The rest six sockets are relative to **servers**, two for the hello/ack server assignment process, two for regular list-sharing communication and the others to ensure presence on the network, and availability. The reasoning behind this division of sockets by underlying context was explained in the Communication Process 1 section. With all these types of different messages arriving, the proxy must be always to receive each kind without blocking the remaining sockets, and for this. **ZeroMQ** provides an API that enables **polling**, checking for events arriving to the polled sockets.

By creating one with the function ***Poller()***, and storing a reference to it in the *poller* attribute, the proxy can then register which sockets must be listening for possible events, by setting the type to *zmq.POLLIN*, to all "receiving" sockets (*backend_r, frontend_r, backend_hello*). It then, as the name implies, polls each one in a non-blocking way, and responds with the appropriate behaviour, which will be now thoroughly delineated.

### 1.3.1 backend_hello and backend_ack

The first polled socket behaves as a "concierge" for the service. It listens for *S_HELLO* messages from the **servers**, and if their ID is not yet registered, it adds it to the list of active connected servers, and also saves it in a queue of "to be added" to the *hash ring*. The reason behind this mechanism stems from a recommendation of the professor, in which it would be bad design if the proxy needed to recalculate / reallocate servers in the ring, every time a new node connected. To address this issue, the group came up with a queue based solution, whereas new servers connect, it only adds them to the ring, when the queue length is equal to **five**, performing the needed operations for correct allocation. By doing this, the system saves on unnecessary operations, which increases performance and availability. Afterwards, it sends an **ACK** message back to the server, to let it know it can start listening for lists, forcing it to enter the listening phase of communication.

### 1.3.2 backend_reach and backend_online

Similar to the ones stated previously, these two sockets handle reaching and acknowledgment from proxy to server and vice versa. However, these are used in every request made from clients, to ensure that the correct node on the ring (position determined by the hashing of its ID), is still online and presentment in the network, available to handle the request. It sends an **S_REACH** message, using a publisher socket, and waits for an **S_ONLINE** confirmation. Afterwards, it simply sends the message to the node, and if needed, replicates the data items to other $N$ neighbour nodes to ensure availability and fault tolerance.

In the case it can't reach the node, the proxy removes it and its position from the ring, including the virtual node's ones.

### 1.3.3 backend_r

This socket is the endpoint for all the **servers** to send their messages. After being requested for lists, and performing all the needed operations (mainly, retrieval and merging), they can just forward the content here and let the **proxy** handle the delivery process to the right clients. This is achieved by taking advantage of the *pub-sub* socket types, where alongside the message (list) the server also sends the ID

of the requesting client list, in a way that only him (and potentially others with the same list on their local storages) receive this latest version of the list. The workflow finishes by reaching the **frontend_s** socket, which is used to forward data to the clients.

#### 1.3.4 frontend_r

The last polled socket represents the messages that arrive from the connected **clients**. Here, as it happens on the previous socket, the proxy verifies the requester's ID, and the message contents. The main component received is the list ID, which by making use of the **Hash Ring** class, the **proxy** will verify if any **server** connected is responsible for the storing process (by comparing the ***hashed key*** to the server's assigned *key ranges*). Afterwards, it must simply forward the message to the correct server, for further operations, by taking once again advantage of the functionality provided by the *pub-sub* ZeroMQ sockets. The workflow finishes by reaching the **backend_s** socket, which is used to forward data to the servers.

# 2 List's Schema

The basis of this project is the dissemination of shopping lists among a network of clients and servers, in a distributed manner. In order to be able to share them, they first need to be well-defined, as in every messaging protocol.

The lists are composed of two main elements, the **version** number, increased with each operation, and an object array, named "list". This array, is made-up of elements that represent the items to acquire, and are composed of an "id" (name / description of the item), "quantity", which specifies how many should be bought, and a *boolean* flag "acquired", which is *False* by default, and is set to *True*, when quantity to acquire reaches zero.

With this definition, all future processes can make use of the specified fields, in order to complete operations successfully.

# 3 ShoppingList CRDT

One of the most important aspects in the scope of the project is the implementation of a **CRDT** (Conflict-Free Replicated Data Type) to manage versioning and inconsistency handling across elements on the network, regarding the distributed data. Since this process could become too daunting for the group members, due to their experience regarding tackling a task of this magnitude, our execution took inspiration, components, and mechanisms found while researching state of the art (simplified) implementations online. Even though the group's version was not developed from scratch, since the basis used was so simple and generic, the changes made to it could account for a productive development process in which it is now perfectly fit for this work's needs and requirements. Without further ado, here is a detailed explanation of the implementation:

## 3.1 Last Writer Wins

The CRDT applied is based on the concept of **Last Writer Wins**, in an *element-set* configuration, to handle list versioning. By assigning a timestamp (or ordinal identifier) to a procedure, we can resolve conflicts that arise based on the results of said concurrent operations.

Alongside the items, a **timestamp** is attached, so that conflicts can be sorted out, during **merge** operations. It takes the form of an **element-set**, which namely contains the *add* and *remove* sets, related to their operation-respective items. By comparing both sets, and if needed, the elements' timestamps, an item can be considered part or not of the merged set. In case of ambiguity, more relevance can be stated to either *add* or *remove* decision bias.

#### 3.1.1 Merge Operation

When divergent versions of the same data are identified by a server or client, a **merge** must resolve said conflict. This is achieved by comparing add ($A$) and remove ($R$) sets, as well as the item's timestamp. If an item is in $A$ and not in $R$, then it's part of the set, if it's in $R$ and not $A$, it's discarded. But if it's in both $A$ and $R$, it is only part of the set if the item's timestamp in $A$ is higher than $R$'s (meaning it was added, removed, then added again). If the same item is in different $A$ sets with different timestamps,

then only the maximum value prevails. The timestamps used need to be coherent and comparable by different nodes in the network, and are generated by the

#### 3.1.2 look_up()

In order to verify the presence of an item in a list, three different checks need to be made: if it's **not** in the *add* set, which means it isn't present, if it is **not** in the *remove* set, which means it is present, or if it's in **both**, where is only present if the timestamp in *add* is newer than in *remove*.

#### 3.1.3 add() and remove()

These two operations are simple and comparable with the last one, where both sets need to be accessed to verify for element presence. When necessary, the timestamp will also be verified, like in the example of add, to see if an item is already in the set with an older timestamp, where only the quantity to acquire will be tallied with the new one.

#### 3.1.4 acquire()

To acquire an item, the system will check (using the CRDT mechanisms) if the specified item (by ID) is in fact in the shopping list, and check its quantity. If the quantity is equal to one, then it decreases to zero, and the boolean attribute *"acquired"*, is set to *True*. If the value is greater than one, it simply decreases. Then, if it is already zero, it raises an exception.

#### 3.1.5 List Convertors

To handle the lists in the system, some conversions during the different phases of the communication process are needed. One method responsible for that, is **convert_to_json_format()**, which by using the **get_full_list()** function, it selects all items from a specified list, and then creates a JSON object according to the **schema** as detailed in the Schema section 2.

Another method is the **save_list_to_file()**, which saves in the client's local machine, in a designated folder, the JSON file containing the list (and elements). Since during development the test were done on a single machine (by each developer), to simulate a multi client environment, the lists were saves in folders named after the client's ID, to mimic and represent their local machines.

# 4 Consistent Hash Ring

The groups' implementation of a *Hash Ring* also takes the form of a *python* class, in which all attributes and methods will be dissected in the following subsections of this paper. When instantiating said class, only two parameters need to be defined: the list of starting servers (nodes) in the ring (which in this setup's case, will be the five started servers by the *service.py* script), and the number of **virtual nodes** to use. While the first is very self-explanatory, the latter needs to be clarified.

In this projects' context, **virtual nodes** refers to the number of **positions** on the ring, which can be interpreted as the logical locations of where data items (lists) can be assigned, based on the value of their hashed key. By default, for each node that is added to the ring, **three** virtual nodes are assigned, all in different positions, since it's calculated using the server's ID (uuid), a slash (-), and the number of the virtual node (1, 2 or 3). By implementing this feature, we enforce the uniform distribution of keys on the space even further, since for a single node, keys that land on three different ranges can be equally assigned. In short, since the service starts with five nodes, with three virtual nodes each, there are **fifteen** different positions data items can land initially.

Other attributes are simple references: *self.ring*, is a dictionary of nodes and associated keys, and *self.sorted_keys*, an array of all keys / positions of the ring, sorted in ascending order, which facilitates readability and usage in methods.

Now that the attributes of this class are well-defined, each of the different **methods** will be thoroughly elucidated.

## 4.1 _hash()

This method presents another functionality that is self-explanatory. It simply takes an argument, and returns the hashed value of provided data. By taking advantage of the **hashlib** python module, it

uses the **md5** algorithm to compute a fixed length, consistent value for the key hash. By hashing the keys, which also represent the positions of data and nodes on the ring, the system guarantees they are uniformly distributed [3], avoiding the creation of certain areas with higher chance of allocating data.

### 4.2 add_node()

In order to have the nodes available for the designated operations (storing, merging), they must be first registered by the service, which is done using this method. Normally they are added five at a time (as explained antecedently), by calling the "ring generator function", however, since this method was developed previously, when the group was adding nodes individually, and it may be needed in the future, for some operation, was left on the codebase, in

### 4.3 remove_node()

This method is the inverse of the previous, since it receives a node ID, and it hashes it according to the virtual node nomenclature *(uuid-(1—2—3))*, to then compare the hash to the ones present on the ring. As it finds the correct node, it simply removes it from the available list. Unfortunately, this process forces the system to recalculate and reallocate keys in the ring, (although only in the affected section, to the next neighbour)to account for partition tolerance and consistency, reducing at the same time, the availability [1].

The second part of this operation involves dealing with reallocation of stored lists. The algorithm finds the next neighbour node from the ring, and selects all data items (lists) present in the node-to-remove storage directory, so it can clone them to the new one's. SInce the removed no longer exits in the ring, all keys that used to fall on its range are now responsibility of the next neighbour, so when a new request arrives to the **proxy**, to get a list, it should be retrieved from the new node.

### 4.4 lookup_node()

To identify the correct destination to deliver the lists, sent by **clients** to **servers**, the **proxy** must compute the hash of the list's key, and find which node is responsible for handling its operations. To do that, this method simply calls the hashing function on said ID, and traverses the list of *sorted* ***node keys***, until it finds one that is larger, which means the list falls within its range (that is defined by the key space from the hash value before the node's, and the anticlockwise neighbour's one). It returns that node's ID.

### 4.5 get_replicas()

Comparing this method with the previous one, not much more can be said, as it is basically the same, as it searches for the responsible node for a data element (key). However, it also finds and returns the previous $N$ (set as parameter) nodes on the ring, by checking the ordered list of positions. This is useful regarding the **replication** aspect, as described in the **Servers** section 1.1, where lists must not only be saved on a node, but also on its $N$ anti-clockwise neighbours. The proxy, then handles routing to said $N$ neighbour nodes

### 4.6 get_responsible_nodes()

This method is present simply for information (and development) purposes. It uses the ring's list of sorted keys, and picks the **first** and **last** values, to define the total keyspace. It cycles through all values, and checks what nodes have them as key, in the ring attribute. It returns a *dictionary* with all nodes, and the respective ranges in keyspace.

### 4.7 print_key_ranges()

The last method present takes full advantage of the last one, to display with extra detail the ranges of keys for each active node in the system, for the same purposes. It shows how many nodes (positions, accounting for virtual nodes) currently exist in the ring, and for each, a list of each of the key ranges (first and last). Since the keys are generated by the ***md5*** algorithm, and can be difficult to "compare", we provided an option set by the **index** parameter, in which, if set to **True**, it returns instead the index of the key in the keyspace (expecting that they are ordered). This allows for an easy understanding

of node allocation, and to verify that distribution is being applied uniformly (each node has the same number of positions, the same distance apart).

# 5 User Interface

Although the processes assigned to the **client role** in the system were already expanded upon in the Clients 1.2 section, they are simply independent methods that still need to be called from somewhere, and that's where the **UI** (user interface) class takes place. Since this project is being developed in an academic context, and the visual aesthetics are not being considered high importance for the work's scope, a minimalist command-line based interface should suffice. The different functionalities from this UI will be dissected forthwith, without going in-depth about the underlying Client class operations, to avoid redundancy.

The *display_menu()* method is called upon class instantiation, alongside the *get_user_choice()* and *process_choice()* for input handling. When the main **menu** is displayed, the possible options are:

1. Create List
2. Print List
3. Add Item to List
4. Remove Item from List
5. Acquire Item
6. Get List From Server
7. Send List to Server
8. Delete List

0. Exit

The first *(1.)* option is the starting point for the life cycle of a list, simply generating an ID (uuid), and creating a new list ***json*** file on the **lists** directory in the client's local machine. Then, so that no empty lists remain on the storage, it is asked to the user to input the data needed to add its first item. The mechanisms to do so are protected against invalid input (e.g. empty strings for the name and non-numeric values for the quantity).

Accessing the more complex remaining options, the second *(2.)*, prompts the user for the desired list's ID, and then displays the *json* object representing the list, with all defined proprieties explained in the **Schema** 2 section.

The *add* and *remove* operations, use the methods present on the **ShoppingList** class, as demonstrated on the **ShoppingList CRDT** section 3. This two options *(3., 4.)* instantiate a new shopping list by using the respective class, and load the list's file to the new object, using the required method. After the fact, and the respective (add or remove) operations execution, the system will try to connect with a **server** through the **proxy**. If it succeeds, then it will retrieve the latest version of the list (with same ID) from the respective responsible node(s), then once again, will take advantage of the shopping list class to **merge** the two lists, and send it back to the same server. If it can't reach the node, it means that the user is in a local/offline environment, so the changes will only be saved locally, and merged when possible [5].

The acquire function and its implementation, represented on the menu by the option five *(5.)* was already expanded upon on the **Last Writer Wins** subsection 3.1 This way, the abstract object can better represent a functional shopping list.

The sixth and seventh options *(6., 7.)* can be compared to *git pull and push* commands, in the context of distributed versioning systems, where all local data (given the underlying context) gets updated to the last and correct version of itself, present on the repository, or sent as the latest version of said data. The first method traverses the (client's) local directory, and for each list present, if one's ID matches the provided by the user, it asks the **proxy** to find and fetch the latest version from the correct server. This way, it is assured the client posses the updated list. The latter, is the opposite, as it sends the specified list by the user (by ID), to the correct and responsible server, so it can be merged, and consequently stored, so that future get operations reflect the most recent changes to data. It's notable that this "send"

method is automatically executed after each data manipulation operation (applied if the client is online), but in the cases where it is not, we allow the user to manually execute it when possible.

The eight *(8.)* option is rather simple, once again, as it simply goes to the client's local directory and traverses it, until it finds the list with the specified ID, removing it from the system

The last *(0.)* option, as it's implied, halts execution and exits the program.

To help visualize different options, and messages shared across entities in the network, a new function was implemented that assigns different colours, based on the first three letters of the message, which corresponds to the identifier of the entities' role (e.g. P¿, S¿, C¿, HR¿). It was also applied to the options in the menu, to improve readability.

# 6 Conclusion

To conclude the project, the group presented a simple approach for an implementation of a collaborative local-first shopping list application, where users can manage different lists on their local machines, and share them with other users for cooperative handling. The developed system was designed with the *"local-first"* philosophy in mind, where users can always access and manipulate data independent of their connectivity status, whilst taking exponential scalability into account.

Considering this last aspect, a versioning management system was employed through a specific implementation of a CRDT, based on the *Last-Writer-Wins* architecture, using the respective (adapted for this project's scope) add, remove, lookup and merge operations.

In terms of distribution, the group used its own implementation of a basic *hash ring*, where nodes were uniformly mapped in a topological ring, each responsible for storing and replicating a list of data keys. Each node on the ring was distributed amongst three different virtual nodes, to further enforce uniformity on positioning. This system is also prepared to tolerate failures, reassigning keys and moving data across the ring, to available neighbours, while updating positions.

*Clients* communicate with *Servers* by using the *ZeroMQ* communication framework, reaching a middleware *Proxy* that handles routing, using a PUB-SUB architecture, based on network entities ID's.

The group is satisfied with the final result and product, since it made each member more familiar with the concepts behind the implemented features, as well a vast array of new technologies and tools that may be used in future work, projects, and ventures.

# References


[1] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012. doi:10.1109/MC.2011.389.

[2] Pieter Hintjens. *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.", 2013.

[3] Anna Pagh and Rasmus Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008.

[4] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types (crdts). *arXiv preprint arXiv:1805.06358*, 2018.

[5] Ink & Switch. Local-first software. you own your data, in spite of the cloud, 2019. URL https://www.inkandswitch.com/local-first/.