A replicated object system

Hayley Patton
hayley@applied-langua.ge

ABSTRACT

We describe *Netfarm*, a replicated object system, in which various kinds of objects can be stored across a network. These objects are instances of schemas (themselves also objects), which describe the representation of objects, and their behaviour, using a portable bytecode. Objects affect each other by running scripts using the bytecode, which in turn produce effects on multi-sets of *computed values*. Arbitrary access to effects is restricted by an *object capability* system on the object scale, and by *capability lists* on a larger scale, allowing untrusted objects to communicate, and for many untrusted applications to run on nodes and client programs. Objects may exist on multiple nodes of the network to make the system fault-tolerant, as their behaviour is explicated to the system and thus reproducible. Replication of the actions generated by scripts cause Netfarm to exhibit *strong eventual consistency*. This programming model allows for supporting many programs on one object system, which can use efficient replicated algorithms and data structures.

CCS CONCEPTS

• **Computing methodologies** → **Distributed programming languages**; • **Computer systems organization** → *Redundancy*; • **Information systems** → *Distributed storage*.

ACM Reference Format:
Hayley Patton. 2021. A replicated object system. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.4712699

1 INTRODUCTION

There is a growing trend of using decentralised networks and systems for online communication and information storage. Such systems have many *servers* or *nodes*, operated and hosted by different parties, and thus do not have central points of failure. Many of these systems exist today, with vastly different applications and network models. These systems can become stifling with their focus on individual applications; while some persistent users may work out how to re-use the infrastructure of a system for another use case, they are invariably going to find difficulties, as they are unable to replace features of the system with their own. Some systems also cannot detect errors and invalid transactions on the system sufficiently, often putting too much trust on a server or node to not misbehave on behalf of the users of that server. For example, some systems may be unable to detect *spoofed* messages, which were not actually sent by their authors, but were sent by someone else with access to the server.

Our initial intentions were to create some sort of fault-tolerant and "accountable" decentralised communication application, and avoid failures caused by server failure and bad operators and users on the network. The system would be accountable in the sense that any computation made could be reproduced, so that a server publishing incorrect results of some computation (perhaps due to a hardware fault, or a malicious operator) could be detected. These intentions are well served by a *distributed* and *replicated* network.

After implementing a basic system with these properties, we found it would be convenient for the system to verify invariants on behalf of an application, and for the system to be capable of storing information for many applications. These goals culminated into a *replicated object system*, in which objects can validate their current states, refusing to be instantiated if they are invalid, and objects can communicate to add "after the fact" references to each other.

1.1 Distribution

We will discuss properties of existing networks in Previous Work, but an abstract understanding of how these networks are formed will be useful.

Many programs and protocols on a network have many clients and one server. Larger systems may use many servers, but usually maintain the illusion that there is only one server somehow. Either form of system is a *centralised* system, as all information is centralised on infrastructure owned by one party. Managing the infrastructure and data is also entirely the duty of that party, which may be unacceptable for some users.

A *decentralised* system partitions such duties to multiple parties. There are two general techniques for decentralisation: federation and distribution. A *federated* system has users pick a server to use as part of a kind of *identity* for information stored on the network. (This server is sometimes called a *homeserver* or *instance*.) An identity provides an unambiguous reference format for any other user of the protocol to retrieve the information. Such an identity usually is made from some pseudo-random string and the name of the server. For example, the Netfarm room on the *Matrix* messaging protocol has the identity of `!YUbZBUZUxCNnDAlxsZ:matrix.org`. The dependence on a server is an immediate drawback of a federated system; that server is the only source of some information, and failure of a server can make the protocol unusable for users of that server.

A *distributed* system avoids this failure mode, as the identity of information is independent of a server on which it is stored. Typically, an identity is instead formed using a cryptographic *hash* of the information; and it can be efficiently searched for using a *distributed hash table*, which provides an algorithm for searching for a node which stores some information. Using a hash to create an identity precludes mutable objects per se, as modifying the state of an object would change its hash. Mutable data must be implemented using a different mechanism, and Netfarm provides *computed values* to simulate mutable data, by computing side effects from the rest of the state of the network.

1.2 Replication

Data on a networked system is either *replicated* or not replicated. In this paper, we will be specific and define a *replicated system* as one where data may be retrieved from multiple sources, and any source is as valid as another. This definition excludes some systems where one server is the *primary source* of some data, and other servers *cache* that data, where only the former server would be able to accept updates to the data. Any data can be retrieved and updated from any server in a replicated system which has a copy of the data.

To our knowledge, replicated systems must be distributed. A centralised system would not be able to replicate, and the server component of object identity in a federated replicated system would have to be useless in order for any server to be able to serve requests for any object. We have not found any non-replicated distributed systems, but replication is a useful property in itself, and sometimes less confusing to discuss (we discuss "distributed" object systems in Previous Work).

Much like decentralised and centralised systems, a replicated system exhibits better fault tolerance than a non-replicated system; if n servers can be used to service a request with a p probability of failing on one server, and failures are independent events, then there is a p^n probability that all servers fail, and a request cannot be serviced. It is thus preferable to have a replicated system, as to minimise the occurrence of faults which render the system inoperable.

However, replicated systems require careful synchronisation and consistency measures, which are further complicated by having to handle node failures, and intentional attempts to manipulate the state of a system. Such measures often lead to difficult-to-understand programs, and programmer experiences not unlike debugging low-level concurrent programs. Netfarm allows a programmer to write replicated systems using a small set of features; but said features can capture the behaviour of many systems, and an object model and message passing can make the resulting program easier to comprehend and more adaptable.

2 PREVIOUS WORK

2.1 Distributed ledgers

2.1.1 Bitcoin. A blockchain creates consensus over the ordering of a transaction log, while allowing for substantial fault tolerance. Typically, blockchains are used to handle digital currencies, or *cryptocurrencies*, but we are more interested in the consensus mechanism and programming techniques they provide. (Although such mechanisms often assume, in part, that there is some currency to compensate honest users with.)

Bitcoin has a simple scripting system for transactions. Scripts are written in a bytecode, which executes on a stack machine. A script is usually used to verify that the user attempting to *spend* some currency is allowed to spend it. The bytecode includes instructions for basic arithmetic, hashing and verifying signatures; but the bytecode is not Turing-complete, nor particuarly expressive. Some non-essential instructions were removed as part of a (fairly lousy) fix for an incorrect implementation of bit-shifting, which would cause nodes to crash sometimes.[1] The bytecode has never supported any looping; it would not be suitable for programming many other uses of the Bitcoin network.

2.1.2 Ethereum. The *Ethereum* network has extended the script system, to allow for clients to program *smart contracts*, which allow for implementations of many complex systems on a blockchain using a Turing-complete language, such as additional currencies, decentralised exchanges, voting systems and some games. The program in a smart contract is executed when it receives a transaction, and the program is described using a bytecode, which has access to a stack, temporary memory in the form of an octet vector, and a permanent map from 256-bit unsigned integers to 256-bit unsigned integers (as described in Chapter 9 of [19]). Contracts can communicate by calling methods of other contracts.

2.1.3 Holochain. The *Holochain* project can be seen as a modification of the blockchain paradigm. Instead of providing consensus on one global transaction log, a user of a Holochain network maintains their own transaction log, and transactions provide cross-references between logs. Verifying a transaction between two parties consists of verifying the logs belonging to either party [5]. By using many separate logs, such a network can scale using the partitioning provided by a *distributed hash table.* (The Ethereum developers intend to improve scaling by *sharding* transactions over 64 smaller chains [8], but nodes in a distributed hash table can scale their contribution to the network at a more fine-grained level.) Scripts for Holochain are written in the *WebAssembly* portable assembler.

2.1.4 Proof of work. Many blockchains use a *proof of work* technique, which makes falsifying the transaction log very computationally expensive [15]. This process requires creating *blocks* of transactions, which will only be accepted by the network if hashes of blocks have sufficiently small values. Such blocks have to be generated by *brute force*, and the act of generating blocks is called *mining*. It is assumed that only an honest majority of users would have the majority of computing power, so an honest network would be able to produce blocks faster than a user attempting to tamper with the order of transactions.

Many parties are interested in dedicating huge computer farms to mining, as they are paid with cryptocurrency when they are first to produce an acceptable block. Unfortunately, a system based on perpetually brute-forcing values, especially at this scale, has the drawback of requiring significant amounts of electrical energy. At the time of writing, [6] estimates the electrical consumption of *Bitcoin* to be somewhat greater than that of the country of Finland. There are other methods of implementing verification for blockchains, but they all require the well-behaving users of a network to have a majority of some scarce resource, such as other currency or storage.

We do not believe we require the strict consistency measures blockchains provide, so we did not use a blockchain to implement Netfarm. However, the ability for clients to program a distributed system can be very useful, so we have taken some inspiration

[1]This is better documented in CVE-2010-5137 and https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2010-5137.

from how blockchains are programmed when implementing our programming system.

2.2 Federated networks

As mentioned before, federated applications delegate data storage and maintaining identity to many servers, although a user will be dependent on one server. In general, cross-server communications only authenticate servers and not users, so servers could falsify messages and state without detection; and while federated systems provide collections of events that are general enough to support multiple forms of presentation, they are still inextensible and cannot support the implementation of new programs.

It can be observed on both systems that large numbers of users are on very few homeservers, allowing much of those networks to be unusable should only those few servers become unavailable.[2] There are many factors influencing the population distribution (which we attempt to examine in the introduction to [2]) but the decentralisation and fault tolerance of federated systems appears to be very questionable.

2.2.1 Matrix. Matrix is a federated chat protocol, which offers a convenient web client, end-to-end encryption, integration with some other protocols (such as IRC and Discord), and integration with some other tools frequently used by its users, such as receiving GitLab and Grafana notifications in a room.

It uses a *room* as an aggregation of *messages*, which is a convenient abstraction for a messaging application, but it is awkward for many other applications. This situation is not unlike, say, implementing the Common Lisp `tagbody` special operator using the condition system. It is certainly possible, but it is strange, and likely inefficient and difficult. Some new applications have been created, such as a script which renders a blog out of the messages in a room;[3] but they do not modify the semantics or structure of Matrix significantly, merely changing the presentation of a room. Instead, we provide a lower-level abstraction, a *computed value* system, but it is capable of implementing rooms (and we will describe how to implement it in *An example use*).

Furthermore, when a user is not using end-to-end encryption, it is possible for a server operator to spoof messages; claiming that a user sent a message which they had not actually sent. Even a simple authorisation mechanism, such as optionally *digitally signing* messages, would be preferable for sending sensitive announcements, where the message cannot be falsified, but it must still be published publicly.

2.2.2 ActivityPub. ActivityPub is a federated micro-blogging protocol, which is used to post short text messages, with optional visual and audio attachments. Well known implementations of ActivityPub include *Mastodon* and *Pleroma*; although it has had many implementations which change the mode of communication somewhat, such as focusing on image or video sharing, or presenting messages in an online forum layout. These deviations from the usual micro-blogging style, however, remain mostly cosmetic, where messages and aggregations of messages are presented to the user in different ways, and different (arbitrary) restrictions are placed on the forms of messages which a server will publish for a user.

2.3 Distributed object systems and Actors

E and Spritely are examples of *distributed object systems*. Object identity is based off server identity in these systems, so it could be argued that distributed object systems of this type are really *federated* by our definitions, but users of these systems call them *distributed* anyway. It follows that they do not replicate objects. There are also separate instructions for sending messages to local objects, which cannot fail, and sending messages to foreign objects, which can fail, as network-related faults can happen.

Erlang may or may not be a distributed object system, depending on who[4] you ask. It is less contentious to call Erlang an *Actor system*. Actors (processes in Erlang parlance) also have node addresses in their identity, but there is only one asynchronous message sending construct [3, p. 66], which can always potentially fail. Again, processes are not replicated in Erlang, but failure detection and replacement is still a design goal of Erlang; which is instead achieved by respawning processes which crash. This sort of fault tolerance is orthogonal to replication, as it generally is used to work around a program generating an invalid state, and not to work around the disappearance of a node.

3 OUR TECHNIQUE

Objects are stored across a distributed hash table, in which it is possible to store objects, retrieve objects by hash, and subscribe to observe *side effects* induced on objects. We provide a class-based object system, where objects are instances of *schemas*, which in turn are objects themselves. These schemas contain lists of slots, *computed slots*, and *scripts* which describe the behaviour of instances of a schema. A schema is an instance of the schema schema, which is an instance of itself (c.f. the Common Lisp metaclass `standard-class`, which is an instance of itself).

All information which a node presents has been verified and/or produced by itself, so nodes do not have to trust each other to make progress when synchronising objects. The faultable nature of the network is hidden from Netfarm objects, as nodes perform all of the computation on behalf of an object. If an object attempts to retrieve another object that the node does not currently store, the computation is paused until that object has been retrieved. We believe this may introduce some storage overhead, as objects that the node is not expected to store will be verified to satisfy requests on behalf of scripts, but this overhead has not been measured.

We use a Kademlia-based distributed hash table [13] to search and retrieve objects, but the design of Netfarm is not dependent on any object storage mechanism. The only requirements of Netfarm on the object storage mechanism are that it must be able to retrieve objects which it would not otherwise retrieve, enabling scripts to

[2]https://social.coop/@Stoori/100542845444542605 shows the usage of ActivityPub servers on a log-log graph. https://the-federation.info/ estimates the total users of the Fediverse to be around 4 million, and the top 12 servers on https://instances.social/list/advanced have around 1.75 million users.
[3]This project is named the "Matrix blog" and is described on https://matrix.org/docs/projects/client/matrix-blog.

[4]…and also when you ask. Joe Armstrong was initially critical of object-oriented programming, leading to an infamous quote starting with "you wanted a banana"; but then his thesis supervisor had convinced him that Erlang was the only really object-oriented language, as it exhibits strong isolation and polymorphism using message passing and independent "object" processes.

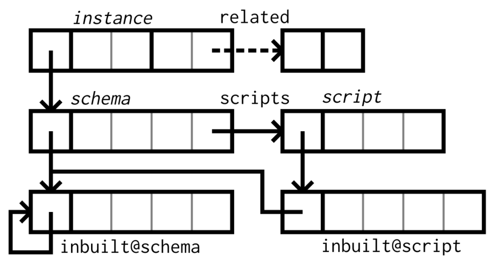

Figure 1: An object is an instance of a schema, which is in turn an instance of the schema schema. We draw objects with box-and-arrow diagrams, using dashed arrows for *computed values*, and labelling arrows with the names of the slots they represent.

retrieve any referenced object on the network, and to provide some metadata about objects for synchronisation.

Objects can be verified using *digital signatures*, associating them with the users that signed them. Such signatures can be used as an authentication mechanism, which allows some actions on objects to be performed only by specific users. (Users are just objects; they do not have any special status in the Netfarm system, except that Netfarm is able to retrieve public keys from user objects.) A *key exchange* mechanism is also provided, using *elliptic-curve Diffie Hellman* (a variant of [7]) keys associated with users of Netfarm, which could be used to encrypt private objects, but it is not used by Netfarm itself.

Objects can contain references to other objects in their slots, which are usually named by hashes of the referenced objects, to create references to other objects (excluding *inbuilt objects*, which are named with symbolic names starting with `inbuilt@`). This would prohibit mutating non-inbuilt objects, as changing the slot values of an object would change its hash, and thus break identity, and it would prohibit creating circular references, as there would be infinite recursion when attempting to compute the hash of any object in the cycle. Side effects are instead caused by running *scripts* which produce *side effects*, adding and removing *computed values* from *computed slots* in objects, which contain multi-sets of values. These computed values allow for "after the fact" references, which do not affect the hash, and thus identity, of an object; so a client will observe the computed slots of the object change.

3.1 Side effects

When designing a system where changes must be replicated over many copies of an object, it is possible that users will observe changes in different orders; so the order in which changes are applied should not affect the result, i.e. the changes are *commutative*, and the data type which is affected by said changes is a *commutative replicated data type* (or CRDT). It is also possible to use a *conflict resolution* mechanism to resolve the ordering of effects, freeing effects of having to be commutative, but we have chosen to use a conflict-free replicated data type, as conflict resolution mechanisms are frequently very specialised, and sometimes hard to get right.

The actions on a computed value system are the addition or removal of a value to a computed slot (herein called a *computed value*). A computed slot is a variation of a *multiset*, which also allows for negative counts of a value, in order for actions to always be commutative. If we disallowed negative counts, for example, removing a value that was not present in a computed value set, then and then adding it would create a different count to adding a value then removing it, and so side effects would not be commutative. The computed value set is a CRDT, which makes Netfarm exhibit *strong eventual consistency*. [17] provides a proof that using CRDTs will provide eventual consistency, so we only need to show how side effects on computed values are commutative, to show that computed values can be used in a system with strong eventual consistency.

THEOREM 3.1. *Actions on computed values are commutative.*

PROOF. Recall that an action on a computed value system is either the addition of a value to a computed slot of an object, or the removal of such a computed value. We could model the state of a computed value system as a mapping of tuples $\langle \text{object}, \text{slot}, \text{value} \rangle$ to an integer *count* of values, each tuple initialized to 0. Adding a computed value causes the count associated with the value to be incremented, and removing a computed value causes the respective count to be decremented. Incrementing and decrementing are special cases of addition ($\lambda x.1 + x$ and $\lambda x.{-1} + x$ respectively), and addition on integers is known to be commutative. Thus our computed value system has commutative actions. □

3.2 Scripts

For side effects to be replicated, we must explicate the behaviour of objects to a Netfarm network. Scripts are run using a virtual machine based on the SECD machine [12] and the Smalltalk interpreter [9]. This virtual machine provides first-class functions and immutable data, and message-passing based polymorphism. The behaviour of objects is aggregated into *scripts*,[5] and schemas have lists of scripts, which are searched to find applicable methods. The behaviour of the virtual machine is entirely deterministic, and thus the complete object system is deterministic.

We decided to use our own virtual machine for a few reasons:

- it is not difficult to write a compiler for a mostly-functional language targeting the SECD machine,
- a virtual machine which exposes some sort of in-memory representation of data, such as WebAssembly, reduces the number of possible techniques that could be used to optimise an implementation of the virtual machine, such as *hash consing* intermediate data (apparently first performed in [10]), and passing host objects into the virtual machine without copying, and
- the virtual machine can enforce various invariants which make cross-object communication significantly less difficult, such as checking function signatures, and ensuring method dispatch works as expected.

[5]The provider of behaviour of an actor in the *Actor model* is sometimes cutely named a *program script*. We did not know of this usage before choosing to call our programs scripts.

A message-passing system appeared to be the simplest system which would enforce *encapsulation*: only the methods of an object can affect the object, a requisite for running multiple programs on one network, without malicious programs being able to clobber the state of other programs. Objects send messages, consisting of a method name and argument list, to other objects, and a receiving object dispatches on the method name. The receiver searches for all methods with the method name defined by scripts, and calls the first applicable method. If a method cannot be found, the process is repeated for a special `does-not-understand` method (much like in Smalltalk), and the method is called with the method name and arguments. Messages can be *authenticated*, as the receiving object can retrieve the sender of a message. If it is necessary to obfuscate the sender of a message, it is not difficult to create a proxy object, which re-sends all messages sent to it to another object.

When a node retrieves a new object, either by replicating it from another node, or by receiving it from a client, it sends an `initialize` message to the object, before it allows other nodes to read its copy of the object. The `initialize` method can thus be used to prevent invalid objects from being created.

Scripts can also induce a partial ordering on the creation of objects in the network. Recall that, if a script attempts to access an object that the node does not yet store, execution of the script is paused. Thus the creation of a referred object always happens before the creation of a referring object.

Using a scripting system with a Turing-complete language and a suitable CRDT could make a replicated system into a sort of "universal simulator" for replicated systems; it is possible to re-implement the core algorithms and data structures of some decentralised systems, including Matrix and Holochain, in Netfarm. However, executing arbitrary untrusted code without any safety measures is a very large security and safety problem; we must provide some boundaries on what programs may do in order to guarantee determinism and safety.

3.2.1 Limiting script misuse. It may be useful to run scripts to present objects and content for clients, and allow clients to interact with objects in a more convenient manner for each object. Conventional "wisdom" surrounding the state of Web browsers, which have run *applets* written in Java, Flash, and now run larger programs written in JavaScript, suggests that this is only a convenient manner to exploit a client's computer, steal their information, or use their computers to perform computing for another party without consent; but we can make these forms of misuse impossible, with little effort.

The behaviour of scripts is restricted by a *capability list*, which prohibits using some runtime features when they are inappropriate; for example, a script run by initializing an object cannot read computed values, as that would allow creating behaviour which is dependent on script execution order, and thus not commutative; but it may write them, and a script run by a client can read computed values, but cannot write computed values. No instructions are ever provided to affect the outside world, and we have implemented the interpreter in safe Common Lisp code, so it should not be possible for a script to escape its environment.

Scripts follow an *object capability model*, where they cannot construct arbitrary references, and can only be given references by other scripts and objects. (This is not the case for users of the Netfarm protocol themselves; replication requires being able to locate all objects, to our knowledge.) Limiting the execution time given to scripts also reduces the computing wastage a malicious script can perform; object initialization must terminate in 300,000 instructions (which can currently be executed in about 5 milliseconds on a relatively new desktop processor), and an interactive client should require significantly fewer than even a million instructions per second to process information for the user; excluding graphical rendering and user input, which are better done in the host environment. There is no situation in which it is possible to perform any significant computation, and then have it transmitted off a client; nor is there a situation where a program could somehow steal confidential information, and then transmit it, nor is there a situation where a network can be misused to perform any significant computation.

4 AN EXAMPLE USE

We have been designing some programs to test the usability of Netfarm. One of these programs is a chat program named *Catfood*,[6] which requires storing messages in a way that allows relatively fast sequential access.

We use an *event stream* data structure, which provides a partial ordering of *events* added to the event stream. Such a stream consists of the stream object, and many event objects which are added to the stream.

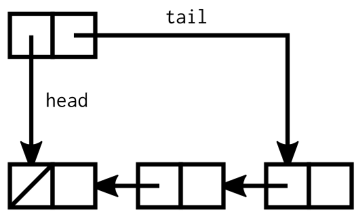

Figure 2: A queue may be constructed using *head* and *tail* references into a linked list.

The event stream resembles how a queue data structure may be constructed from a linked list. (However, we do not remove events from an event stream; we only define adding new events and iterating over all events.) Such a queue would hold references to the first and last conses of the linked list. Similarly, an event stream contains a slot for the first event, and a *wavefront* computed slot. This wavefront provides a fast method to retrieve the latest messages; when an event is created, it is added to the wavefront, and all events observed in the wavefront are removed. Each event contains a slot with the previously observed wavefront, forming a *directed acyclic graph* of events.

With zero latency and no partitioning, this graph will be a linked list, and the wavefront always contains exactly one event. In the presence of latency or a partition, the entire wavefront may not

[6]This program might be used to "eat our own dogfood", should we test it and entrust it with real discussions, but we aren't really dog people. After the bunny, which we used as a logo for Netfarm, the next most loved animal in the early Netfarm community was the cat.

be observed, and only part of the wavefront will be removed by a new event. The true wavefront thus can contain multiple events. Creating another event will return the wavefront to containing only one event. It is not necessary for the wavefront to be consolidated as such, but it will minimise the number of events that a user must sort when iterating over the events of the stream in order.

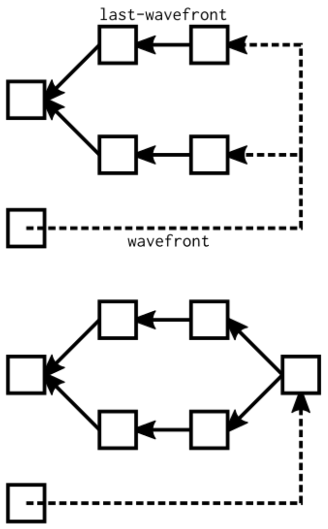

Figure 3: Two writers raced to add events to an event stream. The wavefront contained two events while they were racing; though either writer observed only their last event in the wavefront due to network latency. After they finish racing, another event is added, which stores the previous wavefront, and now the current wavefront contains exactly one event.

Netfarm schemas can be created using a special metaclass. The event class and schema could be defined with:

```
(defclass event ()
  ((last-wavefront :initarg :last-wavefront
                   :reader last-wavefront)
   (stream :initarg :stream :reader stream)
   (successors :computed t :reader successors))
  (:scripts (program-from-local-file "Stream/event.scm"))
  (:metaclass netfarm:netfarm-class))
```

(Note that `program-from-local-file` is defined to compile the source code of a script to a script object, as described later.)

4.1 Searching for a timestamp

Some operations we may require are displaying some of the most recent messages, and displaying messages sent around a given time. These sorts of operations can be performed by traversing the graph, and using a data structure like a *priority queue* to sort events we have not traversed yet, popping them off in order.

One illustrative example is searching for an event at a timestamp; we can work our way back from the wavefront, traversing previous wavefronts, and eventually reach that event. We could choose whether most of the code for this system should be written in Common Lisp (or another host language), or be written to run on the Netfarm script machine. There are advantages to using either language; using Netfarm scripts allow for other users to provide objects subject to some extensions to a protocol, without having to modify a client, but the facilities provided by Common Lisp, including the standard library and compilers[7] may make programming in the host language more favourable.

A function to find an event at a particular timestamp may be defined as:

```
(defun find-timestamp (stream timestamp)
  (labels ((older? (e)      (< (timestamp e) timestamp))
           (newer  (events) (remove-if #'older? events)))
    (loop with q = (make-queue (newer (wavefront stream)))
          until (empty? q)
          do (let ((e (pop-oldest q)))
               (when (= timestamp (timestamp e))
                 (return-from find-timestamp e))
               (insert-all (newer (last-wavefront e)) q)))
    (event-not-found)))
```

The performance of searching could be improved by storing the older wavefronts of previous events, forming a *skip list* of predecessors, allowing faster access to older events to search. The algorithm would otherwise remain the same.

4.2 Programming Netfarm

While a client can be programmed in any mixture of the host language and Netfarm scripts, to replicate side effects and verification of objects, we must write some scripts. These scripts, fortunately, can be compiled from a language which is not as tedious as SECD assembly. We could use the *Slacker* compiler[8], which generates code for our virtual machine from a language which looks superficially like Scheme. The script for an event could be written like:

```
(define (curry f x)
  (lambda (y)
    (f x y)))
(define (map f xs)
  (if (null? xs)
      '()
      (cons (f (car xs))
            (map f (cdr xs)))))

(define (remove-event! s e)
  (call-method s 'remove-from-wavefront e)
  (call-method e 'add-successor (self)))
(define-method (initialize)
  (let ([stream (object-value (self) 'stream)])
    (map (curry remove-event! stream)
         (object-value (self) 'last-wavefront))
    (call-method stream 'add-to-wavefront (self))))
(define-method (add-successor event)
  (add-computed-value 'successors event))
```

[7]We are considering writing a dynamic compiler for Netfarm scripts which would generate Common Lisp code, which would then be compiled to a faster representation using `compile`.
[8]https://gitlab.com/Theemacsshibe/slacker-compiler

Some helper functions (which probably should exist in a standard library) have had to be defined, but they demonstrate that the Netfarm script machine is capable of computing with higher order functions, and returning closures.

The Netfarm implementation represents Netfarm objects as Common Lisp instances, and schemas are translated[9] to Common Lisp classes. It is thus possible to dispatch on the schema of an object from Common Lisp code; the implementation uses this feature to allow for clients to provide methods to duplicate Netfarm slots with specialised representations of Netfarm values, and it has proven useful while testing Netfarm code. During manual testing, we wrote a short method to *present* an event graph visually using the *Common Lisp Interface Manager* [14], (using `format-graph-from-root`) to quickly verify if the graph was being modified as expected.

4.3 Other decentralised log implementations

Our design is mostly based on the *state resolution graph* used in Matrix, described in [1]. Matrix maintains a similar directed acyclic graph of some events, where any event references the power level events which allow it to occur. (Power level events change the *power levels* of users, which are used to grant permissions to them, such as the ability to invite or kick other users.) The events are then topologically sorted to determine all power levels, and then other events can be verified based on the computed levels. The operation of the wavefront (which is also called the set of *forward extremities* in Matrix) is programmed as part of a Matrix server implementation, whereas the event graph in Netfarm is an ordinary application, for which node implementations do not have to have any special implementation code to support.

[11] offers a proof that the Matrix event graph is a CRDT, as well as analysis of how the data structure handles network latency and partitions. However, the implementation in Netfarm is a sort of proof by reduction, as it is possible to rewrite event graph actions into actions on a computed value system, which we have already proven is a CRDT. This proof may be considerably simpler than the aforementioned proof; but it only proves that actions are commutative, not that the data structure works as intended, or is particularly efficient.

[16] is a similar event log, which is implemented atop IPFS, based on a *grow-only set*. This log does not maintain a wavefront or previous wavefronts. Instead, events contain *Lamport timestamps*, where an event contains a timestamp greater than the timestamps of all observed previous events. To access the log sequentially, a list of events and their timestamps is retrieved; which may cause significant space overhead if only retrieving a small subset of events. However, the runtime of sufficiently large accesses will be dominated by the latency of resolving references. A balance between being bottlenecked on network latency, or bottlenecked on network throughput can be made, by allowing the observed wavefront to grow to a relatively large number of events, and only clearing the wavefront when it grows further than that. The previous wavefronts of events would then contain many events, which can be retrieved in parallel.

[9]Schemas themselves are not classes. If that were the case, the class of the schema schema/class would have to be itself, which is not possible to construct in CLOS.

5 CONCLUSIONS AND FURTHER WORK

We have described how we have designed the Netfarm replicated object system, which is notable for providing a programming system, allowing many distributed programs to be implemented on one Netfarm network. We then demonstrated programming with Netfarm by designing a simple event log, demonstrating that it is possible to replicate the behaviour of other decentralised systems on Netfarm easily.

A replicated object system, which has demonstrated itself capable of implementing other decentralised systems, has several benefits. The infrastructure used to implement multiple decentralised systems and networks can be simplfied. It would only be necessary to implement each decentralised system using the object system, and then one network and one node process could be used to host all the systems. An easily accessible network would lower the cost of entry to developing and deploying a new replicated program, as already existing networks and node implementations could be reused. A replicated system designed around objects communicating with *protocols* of messages allows for more extension than in systems with protocols consisting of bare data. Still, some additional features and changes would facilitate using more applications on Netfarm, and would make the programming model provided more understandable.

5.1 Protocols

A sufficiently large distributed system will contain multiple implementations of some concept. These implementations may be abstracted over by programming against protocols. However, it is very likely these implementations are going to use different protocols. [18] bluntly states this problem as "*names* don't scale well [...]. So, [...] we are looking for ways to get things to happen without [...] having to tell any object to 'just go do this'". Names are the least of our problems though; we are more likely to find protocols which have methods with completely different meanings, than the same meanings attached to different names. Some kind of inference strategy may be useful to figure out how to express one protocol in terms of another.

Netfarm does not have an inheritance mechanism, because the techniques used for inheritance can vary wildly. A subtle variation is how a *class precedence list* is computed – Common Lisp uses a simple topological sort for instances of `standard-class`, and Dylan and Python (among other languages) use C3 linearization. A much larger variation is how slots and methods interact; in Common Lisp and Smalltalk (among others), slots and methods are orthogonal and methods cannot override slots, but in Self and Newspeak, methods can override slots. The absence of an inheritance mechanism leaves how to implement inheritance to the user, for better or worse.

5.2 Object identity

Using hashes to form the identity of objects is typical for distributed hash tables, but it can create some strange situations with mutability of any form. It suffices to say that two objects with the same initial state are considered to be the same object, as hashing either object will result in the same hash. This notion of identity can often confuse a programmer, whom is used to creating objects that have the same slot values, but are certainly different; modifying computed values of one object should not make visible changes to another object.

It may be possible to implement a *naming* system, like the system described in [4], where each node names each object with a random but unique identifier specific to each node, and additionally maintains tables of what other nodes have named each object stored. The usual searching used in distributed hash tables could still be employed, by having references contain the usual content hash, as well as the name one node provided. The content hash would be used to locate nodes which likely store the object required, then the name could be translated to retrieve the correct object.

REFERENCES

[1] Neil Alexander. State resolution v2 for the hopelessly unmathematical, 2020. URL https://matrix.org/docs/guides/implementing-stateres.

[2] Applied Language. The Netfarm book, 2020. URL https://cal-coop.gitlab.io/netfarm/documentation/.

[3] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH Royal Institute of Technology, 2003. URL http://erlang.org/download/armstrong_thesis_2003.pdf.

[4] Ganesha Beedubail and Udo Pooch. Naming consistencies in object oriented replicated systems, 1996. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.8209&rep=rep1&type=pdf.

[5] Arthur Brock and Eric Harris-Braun. Holo: Cryptocurrency infrastructure for global scale and stable value, 2017. URL https://files.holo.host/2017/11/Holo-Currency-White-Paper_2017-11-28.pdf.

[6] Cambridge Centre for Alternative Finance. Cambridge Bitcoin electricity consumption index, 2020. URL https://cbeci.org/cbeci/comparisons.

[7] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 1976. ISSN 0018-9448. doi: 10.1109/TIT.1976.1055638.

[8] Ethereum developers. Shard chains, 2021. URL https://ethereum.org/en/eth2/shard-chains/.

[9] Adele Goldberg and David Robson. Smalltalk-80: The language and its implementation, 1983.

[10] Eiichi Goto. Monocopy and associative algorithms in extended Lisp. *University of Tokyo Technical Report TR-74-03*, 1974.

[11] F. Jacob, C. Beer, N. Henze, and H. Hartenstein. Analysis of the Matrix event graph replicated data type. *IEEE Access*, 9:28317–28333, 2021. doi: 10.1109/ACCESS.2021.3058576.

[12] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6 (4):308–320, 01 1964. ISSN 0010-4620. doi: 10.1093/comjnl/6.4.308. URL https://doi.org/10.1093/comjnl/6.4.308.

[13] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric, 2002. URL https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf.

[14] Scott McKay. CLIM: The Common Lisp interface manager. *Commun. ACM*, 34(9): 58–59, September 1991. ISSN 0001-0782. doi: 10.1145/114669.114675.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2007.

[16] OrbitDB. Append-only log CRDT on IPFS, 2016. URL https://github.com/orbitdb/ipfs-log.

[17] Marc Shapiro and Nuno M. Preguiça. Designing a commutative replicated data type. *CoRR*, abs/0710.1784, 2007. URL http://arxiv.org/abs/0710.1784.

[18] Viewpoints Research Institute. STEPS towards the reinvention of programming, 2007. URL http://www.vpri.org/pdf/tr2007008_steps.pdf.

[19] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2020. URL https://ethereum.github.io/yellowpaper/paper.pdf.