HistoryGraph cheap, simple, private, end to end structured data synchronisation

By Mark Lockett

https://twitter.com/marklocketau

https://www.linkedin.com/in/marklockettau/

Published 23rd December 2017

Abstract

The creation of Bitcoin and the blockchain data type has spurred a long dormant interest in decentralised applications. Bitcoin and other cryptocurrencies are decentralised ledgers and while that is a very useful application there are many things we need and want in our lives that are not representable as ledgers.

HistoryGraph allows us to easily create decentralised apps (Dapps) that perform any function imaginable.

HistoryGraph allows 2 or more users to share data by transmitting the history of changes to that data. The history is represented as a hypergraph, the changes are the edges of the hypergraph. As changes are made each user sends their changes to the others.

No processing of these changes performed by servers. Because edges are not processed by servers they can be encrypted, meaning that HistoryGraph can be used to make any app work in an end to end encrypted fashion.

If two users work contemporaneously on the same piece of data the hypergraph will fork. But these forks can merge automatically in all circumstances because HistoryGraph stores data as Conflict Free Replicated Data Types (CRDTs)

Background

HistoryGraph solves five problems with the internet and computers

Privacy

Servers cannot be absolutely safe from spying and interference. User's trust servers but sometimes servers betray us (or their owners or hackers who have taken control of them do). Thanks to Edward Snowden this fact has been well known for several years.

HistoryGraph removes the power of servers by allowing end users to encrypt all communications end to end.

Ease of programming

HistoryGraph makes it possible to write programs without having to write complex Rest APIs or manual data synchronisation management. The program can create a document collection, share it with the user's friends and colleagues and start adding documents to it. The HistoryGraph software will do all the synchronistation automatiocally.

Offline Working

Because we already have a copy of the data, we can still work on it even if the internet is not working. When it comes back on line our document collection will automatically sync with the others.

Performance

When we query our data because it is local the latency is zero. No waiting around for an answer.

Scalability

Because all substantive computations are carried out on the users computer you don't need to write software for servers, ssyadmin servers and worry about providing enough infrastructure to handle large numbers of users.

How does this work?

A user (Abigail) creates a document. At first this will be an empty document, Abigail then adds her own data to this document. This document can be represented as a HistoryGraph (eg a set of changes applied in sequence) . Abigail then sends this set of edges to her friend Bertrand, all of our users already have each other's public keys so this is encrypted end to end.

Abigail continues to work on her document and Bertrand works on the document he just received. After a while both have done work and wish to share back with each other. They both send each other all of the edges they have created. Because Edges can only ever be added to a HistoryGraph both Abigail and Bertrand will now have the same set of edges.

But both of them will have HistoryGraphs with multiple end nodes, but they solve this by creating a Merge edge to merge these two ending together. Merge edges don't change data themselves so both users will create the same merge edges.

They then both replay the HistoryGraph and end up with the same set of Documents and DocumentObjects that the other has and with both users changes merged in together. The Documents and DocumentObjects are gaurantee to be the same in this case because all data is stored as CRDTs.

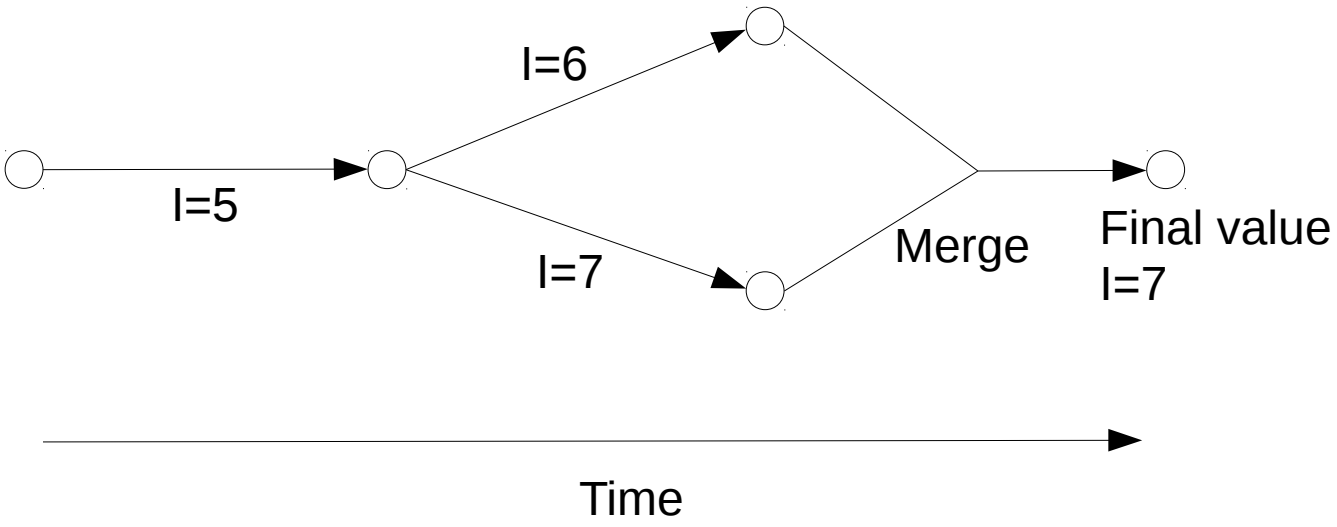
What is it?

The proof of concept (https://github.com/mlockett42/historygraph-poc) program is written in Python. The Proof of Concept demonstrates the kinds of programs that could be built with HistoryGraph. Using HistoryGraph is very similar to programming using an ORM. There is also a separate HistoryGraph library that the proof of concept uses to perform the synchronisation.

HistoryGraph data structure

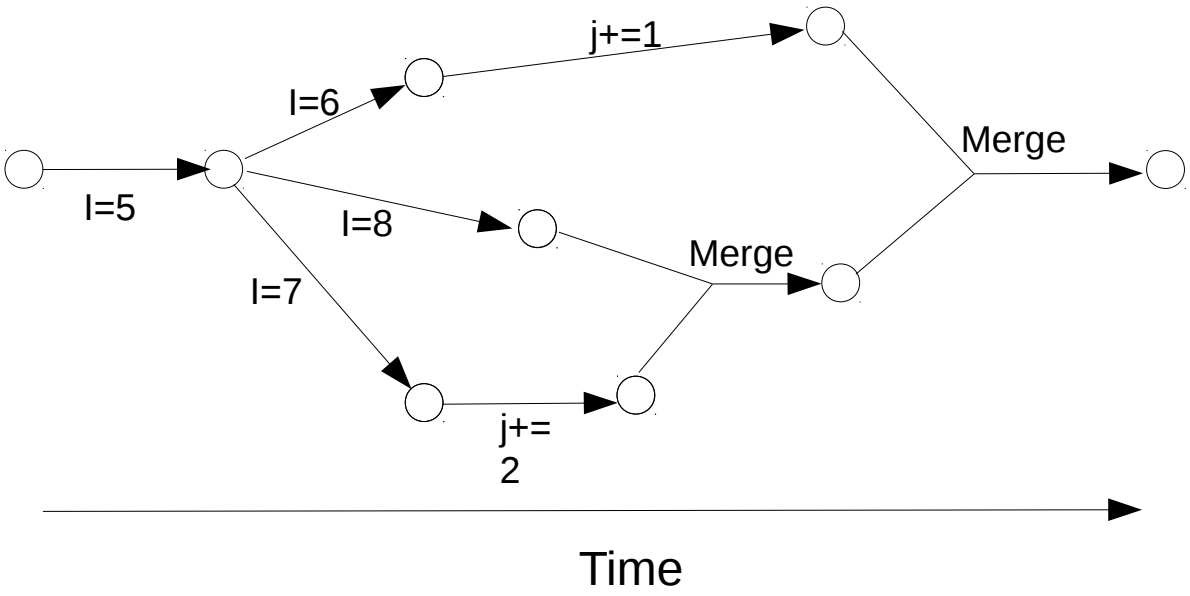
A HistoryGraph is a connected, directed acyclic hypergraph of data changes. It can also be said that a HistoryGraph is a partially ordered set of data changes (a blockchain is a totally ordered set of data changes). In a HistoryGraph a data change is an edge which will have 1 previous edges. In a blockchain each data changes is a block which can have only one previous block. Excepting the first edge or block of course. In HistoryGraph an edge with two previous nodes is called a merge edge, it doesn't actually ever contain a data change of it's own.

Graphically



In the above example, Abigail sets the value of I to 5 then sends her copy of the historygraph to Bertrand. Abigail then changes the value of I to 6 and Bertrand changes the value of I to 7. They then both exchange their full set of edges. When they both have a copy of each others edges they will automatically create a merge edge and compute the final value. The final value of I is 7 in this case. This is because the I=6 and I=7 edges conflict but the higher number wins the conflict resolution in IntRegister data types.

Because CRDTs never conflict graphs like the above can grow to arbitrarily large size and complexity and still converge correctly to the same value. A more complex HistoryGraph might look something like:



In the above example i is an IntRegister and j is an IntCounter. The value of i settles towards 8 because amoungst the conflicting edges the I=8 edge wins. The value of j settles towards 3 because counter edges can never conflict.

HistoryGraphs and Git

If you have ever studied the Git source code control software HistoryGraph should seem familar. The advantage of HistoryGraph being that because all datatypes are conflict free all merges can be performed automatically. Git Commits being equivalent to HistoryGraph Edges.

However HistoryGraph does lack some features Git has commit comments, branchs and tags. Equivalents to these could be added in future releases.

HistoryGraphs and Blockchains

HistoryGraph is a better blockchain, it can fork and remerge. Edges in a HistoryGraph are analogous to transactions in Bitcoin. But because we don't need to pack edges into blocks we don't need to wait for things to settle and can just keep working.

It isn't clear how a crypto-currency could work using a HistoryGraph.

HistoryGraph Library

The HistoryGraph library is similar to an ORM library in Python.

There are four important classes DocumentCollection, Document, DocumentObject and ImmutableObject. DocumentCollection is the equivalent of a database. The intent of DocumentCollection is to represent a single file.

Document - a document contains a set of values that will always be in an internal consistent state. It can contain numeric registers, numeric counters, string registers, lists and collections of DocumentObjects

DocumentObject - same as Document but does do enforce referential integrity.

ImmutableObject – Has member variables which are strings or numbers but cannot be changed after it has been created. The idea is that immutable objects represent data retrieved from nature (eg the temperature at 12noon was 31 degrees). Documents represent data collectively created by humans.

Each of these can have one or more of the following fields as members

Register - a single numerical values. Conflicts are resolved by choosing the greatest number of those in the conflict as the winner.

Counter - Cannot have a conflict the value is the sum of everything added to the counter. Remember we can add negative values.

List - an ordered list of values (specifically DocumentObjects) if two users edit a list contemporaneously it will settle to a consistent value.

Collection - an unordered collection of DocumentObjects this functions as a remove wins set

Text Register - a single string. If there is a conflict the string which is first in alphabetical order wins.

HistoryGraph and Edge objects

The HistoryGraph object belongs to the Document class. If a Document is changed an edge is created and added to that document's HistoryGraph, if a DocumentObject is changed the edge is created and added to the parent or ancestor that is a Document.

Sending and permanent storage of edges and ImmutableObjects is handled by the DocumentCollection. If you don't use the HistoryGraph Communicator and just the HistoryGraph library you will need to implement this yourself although it is pretty easy to do.

Example code

From the Checkers game Proof of Concept (https://github.com/mlockett42/historygraph-poc)

We can define the necessary class just like any class inside a Python ORM (from checkers.py)

```
class CheckersPiece(DocumentObject):
      pieceside = FieldText() # W or B
      piecetype = FieldText() # K for king or blank for pawn
      x = FieldIntRegister()
      y = FieldIntRegister()

class CheckersGame(Document):
      name = FieldText()
      turn = FieldIntCounter() # Even = white's turn odd = black's
      player_w = FieldText()
      player_b = FieldText()
```

```
    pieces = FieldCollection(CheckersPiece)
```

Code is simple python. So the following functions from checkers.py creates the board (comments and debugging checks have been removed)

```
def CreateBoard(self, ll):
    # ll = a list of 8 lists. Each of those lists consists of
    # 8 strings:
    # Blank = no piece "W" or "B" = white or black pawn
    # "WK" or "BK" = white or black king
    y = 0
    for l in ll:
        x = 0
        for s in l:
            if s != "":
                piece = CheckersPiece(None)
                self.pieces.add(piece)
                piece.x = x
                piece.y = y
                piece.pieceside = s[0]
                piece.piecetype = s[1:]
            x = x + 1
        y = y + 1
```

Anyone who understands Python programming with ORMs and Pyside should be able to make a Dapp that gives users full end-to-end encryption.

History Graph Communicator Proof of Concept

The Communicator exists as a proof of concept of how an ecosystem of dapp's based on HistoryGraph might work. It is an email client, it will will opportunistically encrypt emails to other users who it knows use the HistoryGraph communicator. It knows this because the text of each email message contains a message indicating that the emailer is HistoryGraph compatible.

This message also invites any other user to get more information about or even download the HistoryGraph communicator .

The communicator therefore contains a (very crude) email client. Plus three dapps that use the HistoryGraph library to share data peer to peer. The communicator takes care of storage, sending, receiving and encrypting. The three dapps are MultiChat a chat application, a Checkers games and a Trello board application. This is to demonstrate the possiblility of building end to end enrypted social networks, games and small business productivity tools using HistoryGraph technology.

HistoryGraph has no built in technology for establishing trust (unlike Bitcoin for example) it is simply assumed that we will only share documents with other users who we already know. Issues related to this such as man in the middle attacks and end point security will need to be resolved separately.

How do I run the POC?

Instructions

The instructions for how to run this are in the HistoryGraph POC's readme.md file. You have number of options, I would recomend single stepping through the code in IDLE if you want to better understand how the algorithm works.

How do I write an app to run inside the POC?

You can easily build your own app, just inherit from the App class and create a DocumentCollection with all the necessary Documents, DocumentObjects and ImmutableObjects registered inside of it.

For example in the checkers.py file

```
class CheckersApp(App):
	def MessageReceived(s):
		pass


	def CreateNewDocumentCollection(self, dcid):
		dc = super(CheckersApp, self).CreateNewDocumentCollection(dcid)
		dc.Register(CheckersPiece)
		dc.Register(CheckersGame)
		return dc
```

You then need to create a UI for your app. In the proof of concept communicator you just add an item to the appMenu see the file formmain.py. The communicator uses pyside for the user interface.

Note the communicator proof of concept is just a proof of concept. Real HistoryGraph Dapps may end up working in a completely different way.

How do I use the HistoryGraph library independently of the communicator

You simply need to write your own replacements for the functions in the App.py file. You may not need to replace all of them, for example if you don't use ImmutableObjects.

HistoryGraph is also useful for making SPA's two other projects I am working on use this capability binarycrate.com and cavorite.io (both these projects use pypyjs). Binary Crate uses historygraph to automate away the creation of Rest APIs and server side databases. This makes it easier to learn to code.

Explain the HistoryGraph algorithm please

A HistoryGraph is a connected, directed, acyclic hypergraph. It consists of edges, hyper graphs also consist of nodes but in HistoryGraph these are always generatable from replaying edges.

In a HistoryGraph we have a further restrictions above all connect, directed, acyclic hypergraphs

(1) Each edge must have exactly one previous edge (direct predessors) except for Merge edges that can have two previous edges and the first edge in the graph which has zero previous edges. The node preceding the first edge is the least node of the hypergraph and represents a completely empty DocumentObject

(2) There must be only one greatest node in the hyoergraph. If they're are two or more edges without next nodes (direct successors) the HistoryGraph library will build merge edges until there is only one.

User's jointly working on a Document send edges to each other. Adding an edge to a HistoryGraph is a semi-lattice that is a function which is associative, commutative and idempotent. This means that when edges and immutable objects are received it doesn't matter if they are received out of order or more than once. Obviously for any networked application this is a very useful property.

We have a collection of edges and wish to turn these into a collection of Documents and DocumentObjects. This is called (re)playing the HistoryGraph

Step 1

Iterate over the HistoryGraph to make sure there is only one end node. If there is more than one add Merge edges unitl there is only one.

Step 2

Record past edges. We do this because edges which are in each others past can never conflict (only edges that occur contemporaneously can conflict). We iterate over all of the edges and build the set of past edges for them

Step 3

Process conflict winners. Iterate over all edges and then over all edges which are contemporaneous to the first. Edges can only conflict where they are of the same type and changes the same variable. In practice only Registers can have conflicts (FieldCollections are in effect a remove wins set)

We compare the edges that can conflict and determine the winner, the losers are marked as invalid and never run.

Step 4

Replay the edges. Start from the first edge and execute whatever instruction it has. Keep executing the next instrsuction. If we meet an edge with more than one next edge follow one of the strands as far as we can. If we meet a Merge edges only execute it if both previous node have themselves been executed. When we get to end node we are finished and have the Document and DocumentObjects we desire.

The available commands that can be run in a edge are 'Set Property Value' (Register), 'Add Child', 'Remove Child' (FieldCollection), 'Add Int Counter' (FieldCounter), 'Add ListItem', 'Remove ListItem' (FieldList)

What happens next?

This software has lots of issues so I need a small group of programmers to help out. Euthusiasm is the most important quality for this job.

It is necessary first to concentrate on the HistoryGraph library. Some classes and files and concepts within it should be renamed, in some cases their current name does not reflect the graph theory name for the idea and it should.

1) It needs to add Documents to a DocumentCollection upon creation, but this isn't automatically required by the language syntax. But it should be.

2) We need to add DocumentObject to FieldCollections or FieldLists before they are changed this isn't enforced by the API. It will just strangely go wrong if we don't do it.

3) Calling things FieldInt, FieldRegsiter, seems very non-Pythonic and should probably be different

4) It doesn't follow Pythonic naming conventions and probably should.

5) There should be compatible versions in other languages.

6) It does assume that JSON encoding of edges will be the standard but there is no particular reason this would be so.

7) The performance could probably be a lot better, for a start everything is loaded into memory and kept there after playing the edges.

8) Add transactions. Transaction are collections of connected edges, that aren't merge edges. If one edge in a transactions is a conflict losers are of the others should be conflict losers.

9) Add validation, we simply accept whatever our peer sends to us. We could add the ability to validate that the data being sent conforms to business rules. And if it doesn't reject it.

10) Validators and transaction could be combined into a peer-to-peer version of HTTP form validation

11) Migrations: We need some way to migrate the database schema.

12) There are just a lot of code smells. Some of this code was written while I was learning the Python language so it doesn't use language or library feature correctly.

The entire communicator is a proof of concept.

1) For a start there should be mobile version although this could be written using Kivy in the present code base.

2) Think about how programs written in different languages could cooperate in the HistoryGraph app ecosystem

3) The demo apps in the communicator have bugs. The checkers app fails to follow the rules of checkers correctly in some situation (I forget exactly what)

What can I do?

Sign up it keep in touch on the web page.

If you are a coder and want to help join the team.

Send a donation Bitcoin only to: 18c7eZZnWeQhsY7uQCBpHkw5sm1muFVSRn

I if there are sufficient donations I will hire programmers to work on the software

Further Reading

Christopher Miekeljohn has an excellent collection of links about CRDTs at http://christophermeiklejohn.com/crdt/2014/07/22/readings-in-crdts.html