Portobello Road

A secure, distributed, pseudononymous marketplace protocol

Version 638b4a5bf0d944716bc52b098e3623c18c2d6c3e

Portobello Road Working Group

Justin '*J*' Lynn

DRAFT

Key words and phrases. Cryptography, Distributed Marketplace, Overlay Networks

Abstract. The Portobello Road protocol, also known as Portobello Road, enables truly free markets.

Portobello Road is built on a multihub-client model. Hubs are individual servers which may be clustered together or connected into federations of optionally clustered hubs. Clients connect to one or more of these federations and coalesce their available information, initiate transactions and coordinate communications amongst entities in the system.

Communications between entities are protected by a modified OTR messaging protocol which provides for deniable secure communications by which to arrange transactions asynchronously. Because of this, network and network service providers, such as escrow agents, remain ignorant about the true nature of the transactions undertaken by their clients. The only parties with access to the details of the transaction are those entities directly envolved in the transaction of physical goods and services and there is no way for a single participant in the transaction to prove to others after the fact that the other party did indeed create the specified transaction.

Escrow decisions are put directly into the hands of the entities involved by requiring the creation of a Nash equilibrium between them (a technique also known as Mutually Assured Destruction). This allows escrow to be automated and escrow agents are no longer required to be adjudicators in the case of disagreement, only to follow the escrow protocol and to not abscond with the money in transit. If multiparty authorisation is supported by the agreed upon settlement method, escrow agents need only be trusted to follow the escrow protocol – adherance to which is verifiable. The value wagered is not necessarily equal to the value exchanged by parties to the transaction.

Unlike traditional escrow, in this system escrow agents do not remit to a party other than from which the wagered funds were received. Should a transaction be successful, the funds which created the Nash equilibrium between transacting parties are returned to the sending party. Should a transaction fail, the funds are 'destroyed' according to each agent's policy at the direction of either transaction participant – a policy agreed to by all parties. An escrow agent may either choose to keep 'destroyed' funds or not, with the ascent of all parties. Finally, an escrow agent may choose to require direct payment from both, either or neither other party to the transaction. Market forces are expected to decide which mechanism for incentivising escrow agents is more appropriate.

Distribution is acheived via federation. Federation connects hubs in a way in which hub operators may put in place policies which affect and measure the messages flowing between hubs. This permits hub operators the freedom to charge or to not charge for access to their network of merchants and for carrying transaction information between identities. This is effectively how Internet peering works today. Addtionally, hub operators may charge clients for access to their network of merchants and may, as they are no longer needed to provide the escrow functionality centralised marketplaces typically provide, charge merchants for access to their clients. This allows costs to users to reflect the true cost of maintaining the hub system and federation model.

By these mechanisms, Portobello Road provides for a need-to-know market in which buyers and sellers may transact, escrow and settle freely and without coercion. Peer-to-peer review and attestation provide for reputation and decentralised escrow protects against malicious entities while allowing individuals to settle disputes in the manor they choose. Users are not coerced into staying with one organisation or service provider but are free to choose and take their data and business to the best providers where best is defined by them and no one else.

DRAFT

Contents

DRAFT

DRAFT

DRAFT

DRAFT

Foreword

The open society, the unrestricted access to knowledge, the unplanned and uninhibited association of men for its furtherance — these are what may make a vast, complex, ever growing, ever changing, ever more specialized and expert technological world, nevertheless a world of human community.

Robert Oppenheimer
Science and the Common Understanding
1953

DRAFT

DRAFT

Conventions

The keywords *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in this document are to be interpreted as described in RFC2119 (see full text of RFC on page 77).

DRAFT

DRAFT

Part 1

Architecture

DRAFT

DRAFT

CHAPTER 1

Identities

1. Attributes

2. Storage

3. Function

3.1. Attestation.

3.2. Messaging.

3.3. Escrow.

3.4. Settlement.

DRAFT

DRAFT

CHAPTER 2

Messages

DRAFT

DRAFT

CHAPTER 3

Marketplaces

DRAFT

DRAFT

Part 2

Components

DRAFT

DRAFT

CHAPTER 4

Data Model

DRAFT

DRAFT

CHAPTER 5

Message Types

1. Relationships

1.1. Attestation.

1.2. Deattestation.

1.3. External Network Transaction Reference.

2. Offerings

2.1. Item.

2.2. Auction.

2.3. Service.

2.3.1. *Escrow.*

2.4. Suspension.

2.5. Withdrawl.

3. Communications

3.1. Offer Queries.

3.1.1. *Offers Available.*

3.1.2. *Set of Offers.*

3.2. Identity Queries.

3.2.1. *Get Identity.*

3.2.2. *Set of Identities.*

3.3. Propositions.

3.3.1. *Proposition.*

3.3.2. *Counter Proposition.*

3.3.3. *Acceptance.*

3.3.4. *Denial.*

3.4. Inquiry.

3.4.1. *Inquiry.*

3.4.2. *Public Inquiry.*

3.4.3. *Response.*

DRAFT

DRAFT

CHAPTER 6

Message Formats

DRAFT

DRAFT

CHAPTER 7

Marketplaces

1. Hub

2. Federation

DRAFT

DRAFT

CHAPTER 8

Transaction Distribution

DRAFT

DRAFT

CHAPTER 9

Client

DRAFT

DRAFT

Part 3

Higher Level Composition

DRAFT

DRAFT

CHAPTER 10

Attestation

Associating an identity with another identity. Typically in the real world with the help of another identity that is trusted to tell the truth.

What is attestation? Effectively, an assertion of review.

1. Functional Necessity

1.1. Federation.

1.1.1. *Authentication.*

1.1.2. *Management.*

1.1.3. *Policy Voting and Enactment.*

1.2. Declaration of Association.

1.2.1. *Interest Groups.*

1.2.2. *Reviews and Ratings.*

1.2.3. *Approval or Disapproval.*

1.2.4. *Friend or Foe.*

1.2.5. *Governmental or Organisational Association (e.g. Citizenship, Residency, Employment).*

1.3. Regulatory Compliance.

1.3.1. *Geopolitical Restriction.*

1.3.2. *Customs Requirements.*

1.3.3. *Approved Buyers Lists.*

1.3.4. *Restricted Materials Lists.*

1.4. Local Nature of Attestation Enforcement.

2. Benefits

2.1. Buyers.

2.2. Marketplaces.

2.3. Merchants.

3. Creation

4. Types

4.1. Positive.

4.2. Negative.

4.3. Informational.

DRAFT

5. Revocation

DRAFT

CHAPTER 11

Federation

1. Mechanism

1.1. Creation.

1.2. Membership.

1.2.1. *Initial Attestation of Outside Identities.*
1.2.2. *Attestation of Hubs Controlled by Attested Identity.*
1.2.3. *Deattestation of Hubs Controlled by Attested Identity.*
1.2.4. *Majority Deattestation of Hubs Controlled by Attested Identity.*
1.2.5. *Deattestation of Attested Identity.*
1.2.6. *Majority Deattestation of Attested Identity.*

1.3. Destruction.

1.4. Message Storage.

1.4.1. *Disconnected Members.*

1.5. Message Replication.

1.5.1. *Compact Representation of Changed State.*
1.5.2. *Binary Asset Update.*

1.6. Message Routing.

1.6.1. *Optimal Message Distribution.*
1.6.2. *Loop Prevention.*

2. Statistics

3. Reporting

4. Policy

4.1. Settlement.

4.1.1. *Traffic Oriented.*
4.1.2. *Network Value Oriented.*

4.2. Content.

4.2.1. *Attestation.*
4.2.2. *Descriptive Filtering.*
4.2.3. *Manual Review.*

DRAFT

DRAFT

CHAPTER 12

Offering

DRAFT

DRAFT

CHAPTER 13

Discovery of Offering

1. Search

2. Ontology

3. Recommendation

DRAFT

DRAFT

CHAPTER 14

Interidentity Communication

1. Public Questions and Answers

1.1. Related to an Offering.

1.2. Related to an Identity.

2. Private Messaging

2.1. Purchase Status Updates.

DRAFT

DRAFT

CHAPTER 15

Taking an Offering

DRAFT

DRAFT

CHAPTER 16

Escrow

1. Agent Identity

2. The Problem of Trust

3. Nash Equilibrium Solution

DRAFT

DRAFT

CHAPTER 17

Settlement

DRAFT

DRAFT

Part 4

Conformance

DRAFT

DRAFT

CHAPTER 18

Hub

DRAFT

DRAFT

CHAPTER 19

Client

DRAFT

DRAFT

Part 5

Reference Implementation

DRAFT

But when you come right down to it the reason that we did this job is because it was an organic necessity. If you are a scientist you cannot stop such a thing. If you are a scientist you believe that it is good to find out how the world works; that it is good to find out what the realities are; that it is good to turn over to mankind at large the greatest possible power to control the world and to deal with it according to its lights and its values.

Robert Oppenheimer
2 Nov 1945

DRAFT

CHAPTER 20

Barrow Lib JS

1. Model

barrow-lib-js-model
Primary Barrow Lib JS high-level interface.

1.1. Directories.

1.2. Marketplaces.

1.3. Connections.

1.4. Identities.

1.5. Attestation.

1.6. Views.

1.7. Listings.

1.8. Transactions.

1.9. Orders.

2. Store

barrow-lib-js-store

2.1. Account and Key Material Storage. barrow-lib-js-store-keys
Data store for encrypted content. Peer-Assisted Key Derivation (http://justmoon.github.io/pakdf/) support?

2.2. Key/Content Addressable Storage. barrow-lib-js-store-cas
General Data Store. Encrypted, authorisation compatible storage.

2.3. Time Ordered Storage. barrow-lib-js-store-timeordered
What happened when in what order. Useful for auctions.

3. Identify

barrow-lib-js-identify

3.1. Identity. barrow-lib-js-identify-identity

3.2. Authenticate. barrow-lib-js-identify-authenticate

3.3. Attest. barrow-lib-js-identify-attest

DRAFT

4. Communicate

barrow-lib-js-communicate

4.1. EnDec. barrow-lib-js-communicate-endec
Encrypt/Decrypt

5. Browse

barrow-lib-js-browse

6. List

barrow-lib-js-list

6.1. Listing Types.
6.1.1. *Immediate.* barrow-lib-js-list-immediate
6.1.2. *Auction.* barrow-lib-js-list-auction

7. Transact

barrow-lib-js-transact

8. Settle

barrow-lib-js-settle

8.1. Escrow. barrow-lib-js-settle-escrow

8.2. Payment. barrow-lib-js-settle-payment
8.2.1. *Ripple.* barrow-lib-js-settle-payment-ripple

DRAFT

CHAPTER 21

Barrow

Server daemon which implements storage, search, federation and message routing – known as a Hub in the protocol. Implemented in CoffeeScript, primarily, with some JavaScript components. CoffeeScript is a language that is compiled to JavaScript.

Barrow ships a command-line interface only. End user configuration is accomplished via Barrow Client.

DRAFT

DRAFT

CHAPTER 22

MerchantD

DRAFT

DRAFT

CHAPTER 23

EscrowD

DRAFT

DRAFT

CHAPTER 24

Barrow Client

A Barrow client implemented in CoffeeScript, with some JavaScript components. The client and server components are both implemented in the same language and using the same common components. Barrow Client imports components from Barrow Lib JS, described on page 59.

DRAFT

DRAFT

CHAPTER 25

Barrow Client Desktop

A client styled after the Tor Browser Bundle, downloadable. Verified and signed or buildable from source. Solves the chicken-and-egg problem of client-side javascript crypto.

DRAFT

DRAFT

Part 6

Addenda

DRAFT

DRAFT

Colophon

This document is typeset with LaTeXand the pdf document is rendered with pdfLaTeX. The latest LaTeXsource for this document and its associated program code is found at `https://github.com/portobello-road/whitepaper`. Version control is accomplished with the Git Version Control System using the git flow workflow. Source Code Syntax Highlighting accomplished with the 'minted' package. The 'minted' package utilises the pygments source code syntax highlighting engine and is configured to use the 'tango' syntax highlighting style. Psuedocode typesetting is accomplished via the 'algorithmicx' package.

DRAFT

DRAFT

Part 7

Appendices

DRAFT

DRAFT

APPENDIX A

RFC2119 — Key words for use in RFCs to Indicate Requirement Levels

DRAFT

Network Working Group S. Bradner
Request for Comments: 2119 Harvard University
BCP: 14 March 1997
Category: Best Current Practice

Key words for use in RFCs to Indicate Requirement Levels

Status of this Memo

This document specifies an Internet Best Current Practices for the Internet Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Abstract

In many standards track documents several words are used to signify the requirements in the specification. These words are often capitalized. This document defines these words as they should be interpreted in IETF documents. Authors who follow these guidelines should incorporate this phrase near the beginning of their document:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Note that the force of these words is modified by the requirement level of the document in which they are used.

1. MUST This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

2. MUST NOT This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

3. SHOULD This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

4. SHOULD NOT This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

5. MAY This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option

DRAFT

```
   MUST be prepared to interoperate with another implementation which
   does not include the option (except, of course, for the feature the
   option provides.)

6. Guidance in the use of these Imperatives
   Imperatives of the type defined in this memo must be used with care
   and sparingly.  In particular, they MUST only be used where it is
   actually required for interoperation or to limit behavior which has
   potential for causing harm (e.g., limiting retransmisssions)  For
   example, they must not be used to try to impose a particular method
   on implementors where the method is not required for
   interoperability.

7. Security Considerations
   These terms are frequently used to specify behavior with security
   implications.  The effects on security of not implementing a MUST or
   SHOULD, or doing something the specification says MUST NOT or SHOULD
   NOT be done may be very subtle. Document authors should take the time
   to elaborate the security implications of not following
   recommendations or requirements as most implementors will not have
   had the benefit of the experience and discussion that produced the
   specification.

8. Acknowledgments
   The definitions of these terms are an amalgam of definitions taken
   from a number of RFCs.  In addition, suggestions have been
   incorporated from a number of people including Robert Ullmann, Thomas
   Narten, Neal McBurnett, and Robert Elz.

9. Author's Address
      Scott Bradner
      Harvard University
      1350 Mass. Ave.
      Cambridge, MA 02138
      phone - +1 617 495 3864
      email - sob@harvard.edu
```

DRAFT

DRAFT

APPENDIX B

Semantic Versioning Standard – Version 2.0.0-rc.1

In the world of software management there exists a dread place called "dependency hell." The bigger your system grows and the more packages you integrate into your software, the more likely you are to find yourself, one day, in this pit of despair.

In systems with many dependencies, releasing new package versions can quickly become a nightmare. If the dependency specifications are too tight, you are in danger of version lock (the inability to upgrade a package without having to release new versions of every dependent package). If dependencies are specified too loosely, you will inevitably be bitten by version promiscuity (assuming compatibility with more future versions than is reasonable). Dependency hell is where you are when version lock and/or version promiscuity prevent you from easily and safely moving your project forward.

As a solution to this problem, I propose a simple set of rules and requirements that dictate how version numbers are assigned and incremented. For this system to work, you first need to declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once you identify your public API, you communicate changes to it with specific increments to your version number. Consider a version format of X.Y.Z (Major.Minor.Patch). Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version.

I call this system "Semantic Versioning." Under this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next.

DRAFT

1. Semantic Versioning Specification (*SemVer*)

(1) Software using Semantic Versioning MUST declare a public API. This API could be declared in the code itself or exist strictly in documentation. However it is done, it should be precise and comprehensive.
(2) A normal version number MUST take the form X.Y.Z where X, Y, and Z are non-negative integers. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically by increments of one. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.
(3) When a major version number is incremented, the minor version and patch version MUST be reset to zero. When a minor version number is incremented, the patch version MUST be reset to zero. For instance: 1.1.3 -> 2.0.0 and 2.1.7 -> 2.2.0.
(4) Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications must be released as a new version.

(5) Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable.
(6) Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.
(7) Patch version Z (x.y.Z | x -> 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
(8) Minor version Y (x.Y.z | x -> 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.
(9) Major version X (X.y.z | X -> 0) MUST be incremented if any backwards incompatible changes are introduced to the public API. It MAY include minor and patch level changes. Patch and minor version MUST be reset to 0 when major version is incremented.
(10) A pre-release version MAY be denoted by appending a dash and a series of dot separated identifiers immediately following the patch version. Identifiers MUST be comprised of only ASCII alphanumerics and dash [0-9A-Za-z-]. Pre-release versions satisfy but have a lower precedence than the associated normal version. Examples: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.
(11) A build version MAY be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch version or pre-release version. Identifiers MUST be comprised of only ASCII alphanumerics and dash [0-9A-Za-z-]. Build versions satisfy and have a higher precedence than the associated normal version. Examples: 1.0.0+build.1, 1.3.7+build.11.e0f985a.
(12) Precedence MUST be calculated by separating the version into major, minor, patch, pre-release, and build identifiers in that order. Major, minor, and patch versions are always compared numerically. Pre-release and build version precedence MUST be determined by comparing each dot separated identifier as follows: identifiers consisting of only digits are compared numerically and identifiers with letters or dashes are compared lexically in ASCII sort order. Numeric identifiers always have lower precedence than non-numeric identifiers. Example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0-rc.1+build.1 < 1.0.0 < 1.0.0+0.3.7 < 1.3.7+build < 1.3.7+build.2.b8f12d7 < 1.3.7+build.11.e0f985a.

DRAFT

2. Why Use Semantic Versioning?

This is not a new or revolutionary idea. In fact, you probably do something close to this already. The problem is that "close" isn't good enough. Without compliance to some sort of formal specification, version numbers are essentially useless for dependency management. By giving a name and clear definition to the above ideas, it becomes easy to communicate your intentions to the users of your software. Once these intentions are clear, flexible (but not too flexible) dependency specifications can finally be made.

A simple example will demonstrate how Semantic Versioning can make dependency hell a thing of the past. Consider a library called "Firetruck." It requires a Semantically Versioned package named "Ladder." At the time that Firetruck is created, Ladder is at version 3.1.0. Since Firetruck uses some functionality that was first introduced in 3.1.0, you can safely specify the

Ladder dependency as greater than or equal to 3.1.0 but less than 4.0.0. Now, when Ladder version 3.1.1 and 3.2.0 become available, you can release them to your package management system and know that they will be compatible with existing dependent software.

As a responsible developer you will, of course, want to verify that any package upgrades function as advertised. The real world is a messy place; there's nothing we can do about that but be vigilant. What you can do is let Semantic Versioning provide you with a sane way to release and upgrade packages without having to roll new versions of dependent packages, saving you time and hassle.

If all of this sounds desirable, all you need to do to start using Semantic Versioning is to declare that you are doing so and then follow the rules. Link to this website from your README so others know the rules and can benefit from them.

3. FAQ

3.1. How should I deal with revisions in the 0.y.z initial development phase?

The simplest thing to do is start your initial development release at 0.1.0 and then increment the minor version for each subsequent release.

3.2. How do I know when to release 1.0.0?

If your software is being used in production, it should probably already be 1.0.0. If you have a stable API on which users have come to depend, you should be 1.0.0. If you're worrying a lot about backwards compatibility, you should probably already be 1.0.0.

3.3. Doesn't this discourage rapid development and fast iteration?

Major version zero is all about rapid development. If you're changing the API every day you should either still be in version 0.x.x or on a separate development branch working on the next major version.

3.4. If even the tiniest backwards incompatible changes to the public API require a major version bump, won't I end up at version 42.0.0 very rapidly?

This is a question of responsible development and foresight. Incompatible changes should not be introduced lightly to software that has a lot of dependent code. The cost that must be incurred to upgrade can be significant. Having to bump major versions to release incompatible changes means you'll think through the impact of your changes, and evaluate the cost/benefit ratio involved.

3.5. Documenting the entire public API is too much work!

It is your responsibility as a professional developer to properly document software that is intended for use by others. Managing software complexity is a hugely important part of keeping a project efficient, and that's hard to do if nobody knows how to use your software, or what methods are safe to call. In the long run, Semantic Versioning, and the insistence on a well defined public API can keep everyone and everything running smoothly.

3.6. What do I do if I accidentally release a backwards incompatible change as a minor version?

As soon as you realize that you've broken the Semantic Versioning spec, fix the problem and release a new minor version that corrects the problem and restores backwards compatibility. Remember, it is unacceptable to modify versioned releases, even under this circumstance. If it's appropriate, document the offending version and inform your users of the problem so that they are aware of the offending version.

DRAFT

3.7. What should I do if I update my own dependencies without changing the public API?.

That would be considered compatible since it does not affect the public API. Software that explicitly depends on the same dependencies as your package should have their own dependency specifications and the author will notice any conflicts. Determining whether the change is a patch level or minor level modification depends on whether you updated your dependencies in order to fix a bug or introduce new functionality. I would usually expect additional code for the latter instance, in which case it's obviously a minor level increment.

3.8. What should I do if the bug that is being fixed returns the code to being compliant with the public API (i.e. the code was incorrectly out of sync with the public API documentation)?

Use your best judgment. If you have a huge audience that will be drastically impacted by changing the behavior back to what the public API intended, then it may be best to perform a major version release, even though the fix could strictly be considered a patch release. Remember, Semantic Versioning is all about conveying meaning by how the version number changes. If these changes are important to your users, use the version number to inform them.

3.9. How should I handle deprecating functionality?

Deprecating existing functionality is a normal part of software development and is often required to make forward progress. When you deprecate part of your public API, you should do two things:

(1) update your documentation to let users know about the change
(2) issue a new minor release with the deprecation in place. Before you completely remove the functionality in a new major release there should be at least one minor release that contains the deprecation so that users can smoothly transition to the new API.

4. About

The Semantic Versioning specification is authored by Tom Preston-Werner, inventor of Gravatars and cofounder of GitHub.

If you'd like to leave feedback, please open an issue on GitHub.

5. License

Creative Commons - CC BY 3.0 http://creativecommons.org/licenses/by/3.0/.

DRAFT