Monetization on the Modern Web: Automated Micropayments From Bitcoin-Enabled Browsers

Samvit Jain and Arvind Narayanan
Department of Computer Science, Princeton University

Abstract

In this study, we propose and evaluate a software implementation of a Bitcoin micropayments-based revenue system for online businesses, which enables users to make small payments to access web content on a per-use basis, in lieu of viewing ads or signing up for a credit card subscription. We focus in particular on resolving a known issue with past conceptions of micropayment systems, namely that asking users to repeatedly make payment decisions about online content they have not yet experienced imposes a cognitive load, deterring usage. Our solution takes the form of a client-side browser extension, which handles the logistics of making payments, via the use of special HTTP header fields and integration with a client's Bitcoin wallet, but also automates the decision process, by taking appropriate action based on a user's previously indicated preferences. Our system succeeds in eliminating any extra, payment-related actions from the process of browsing the web, a significant step toward removing the mental transactions costs associated with micropayments. We conclude by evaluating our software on the basis of various other criteria, such as ease of installation, security, and scalability, to illustrate avenues for future work in the area.

1 Introduction

Loading a 2000-word CNN article on a mobile phone involves over 200 HTTP requests to 25 different domains, uses around 2MB of mobile data, and takes 13 seconds on an average 4G LTE network [1]. This is the state of the art in online monetization; the giveaway signs of a massive consortium of ad scripts and tracking devices aiming to convert a user's web activity into actionable insights, and derive revenue from free content.

The numbers belie the shortcomings. Advertising infrastructure forms the bottleneck in web page performance, adding between 4 and 12 seconds to the load times of 25 top news sites in the U.S [2]. Advertising as a business model necessitates third-party tracking, and the aggregation of consumer data across disparate sources, an infringement on user privacy [3]. Ads themselves clutter websites and compete with the primary content of a page for a user's attention [4]. Given these problems, it is unsurprising that advertising is under attack – from digital rights initiatives such as Do Not Track, from companies building new products to help users evade ads, and from the proliferation of content blockers on desktop and mobile devices.

The other leading form of content monetization on the web, the subscription model, suffers from its own host of issues. To sign up for a subscription requires a user to provide a credit card number and a billing address, two of the most sensitive pieces of personal information. Moreover, the process of subscribing itself entails a fixed time cost, making the idea of holding many subscriptions, even if they were financially inexpensive, infeasible [5]. The final nail in the coffin in the psychology of paywalls. Paywalls stem impulse usage, conditioned as users are to avoiding them, and turn away infrequent visitors, who might have been customers under a more flexible pricing scheme [6].

1.1 Micropayments

Micropayments, the concept of making payments to a website on a per-use basis, offer a promising alternative to traditional content monetization schemes. By asking users to pay for access to an online service, but in a non-

contractual way, the micropayments model combines the service quality expectations associated with subscriptions with the accessibility of free content. The fact that users, not advertisers, now subsidize web content obviates some of the privacy and performance issues discussed earlier; unlike with advertising, little additional infrastructure, beyond that used to collect usage data for analytics purposes, is needed to track and bill a visitor for usage. Bitcoin-based micropayments, moreover, enable users to pay for content without providing highly sensitive information, or entering into long-standing contracts. The result is a flexible, granular revenue model that incentivizes high-quality content and attention to in-site experience.

Previous conceptions of micropayments, however, have suffered from a problem known as cognitive load [7]. The assumed implementation generally involves a "click to pay" model, in which a users clicks on a button to pay a site a fixed fee as a prerequisite to access [8]. This design forces users to place a value on a good they have not yet experienced, and take an explicit, separate action to pay for it. Moreover, the mental transaction costs associated with these decisions do not fall linearly with the price of the good; namely, purchasing an online service worth $0.10 does not take just one-tenth of the mental effort involved in purchasing one worth $1.00 [9]. This cognitive load adds up over the course of a browsing session, deterring usage and resulting in less purchased value than under a flat-fee, or bundled, pricing model [10, 11]. Micropayments have thus been dismissed by many technologists as a fundamentally flawed proposition, the alleged problem being psychological and economic in nature, not a matter of poor implementation [10].

1.2 Contributions

In this paper, we argue that a technical implementation that deliberately abstracts away small decisions, combined with a correctly designed permission model, can in fact alleviate the problem of cognitive load. Our proposed solution, an automated system for funding a user's Internet activity, invokes the use of HTTP headers to communicate billing schemes and proofs-of-payment, and a front-loaded permissions model, to allow a client-side browser extension to make payment decisions on behalf of a user.

Together, these components ensure that users do not have to take discrete actions on each page view. Given the permission to spend at any rate below a threshold indicated on install, the extension can determine whether a billing scheme is consistent with a user's willingness to pay, and take action accordingly. Furthermore, the use of HTTP headers to transmit payment instructions and proofs-of-payment enables the request-response logic for payments to be built into the Javascript of a webpage itself. This frees the client from the responsibility of initiating and maintaining a separate connection with the server to make payments, leaving to it only the task of adding HTTP headers to pre-scheduled GET requests made by the webpage. Finally, by allowing a user to start or stop a payment stream by simply opening or closing a browser tab, respectively, we tie our implementation of micropayments to natural actions a user makes while browsing the web.

2 Background

2.1 Bitcoin

We begin by motivating our decision to use Bitcoin in our implementation of micropayments. The key observation is that traditional payment mechanisms degrade poorly in the limit of small transaction values, due to the high fixed costs associated with enabling trustless transactions on the web. Current approaches to online payments involve either hypersensitivity to fraud, with a trusted third-party blocking all transactions that look remotely suspicious (the PayPal model) [12], or make the assumption that the expected reputational damage to a vendor that attempts to cheat or overcharge a client will exceed the gains from strategic default (the credit card model). The former entails obvious deadweight losses, from foregone legitimate transactions, and requires a central authority to mitigate every transaction; the later involves revealing a credit card number (i.e. a private key) to an unknown entity on the web to make payments of any value, an alarming concept to anyone familiar with public-key cryptography. In particular, when the payment values are small, fraud mitigation costs begin to dominate the transaction, and the reputational risk assumption breaks down.[1] Indeed, fixed costs are part of the reason why PayPal charges sellers a min-

[1] Far fewer websites have the reputation to credibly ask users for credit card numbers than have the ability to offer web content at low prices.

imum fee of $0.30, over the 2.9% cut they take on each transaction [13].

Standard Bitcoin transactions in fact also entail prohibitive transaction fees. To ensure that a Bitcoin transaction is incorporated in the blockchain, the global ledger that establishes the legitimacy of Bitcoin payments, the issuer of the transaction must include an incentive, in the form of a transaction fee, for Bitcoin miners, the entities responsible for updating the blockchain. These fees vary depending on the output values of the transactions (which roughly signal how "important" a transaction is) and the size in bytes of the transaction [14]. As of April 2016, to ensure that a transaction is incorporated in the blockchain relatively quickly entails a transaction fee of 40 Satoshis per byte, or 9,040 Satoshis ($0.04) for the median transaction size of 226 bytes [15].

2.2 Micropayment Channels

The workaround, and the prerequisite for any cost-effective implementation of Bitcoin micropayments, is a client-server contract known as micropayment channels. The channels protocol allows a client to make a series of payments to a server of very fine granularity (i.e. down to 1 Satoshi, or .0004 cents), while only publishing and paying transaction fees on the first and last transactions [16]. Using this protocol, the server can provide a client access to a resource, such as a multi-page news article or a content feed, in an incremental way, and accept metered payments for doing so.

The mechanics of the protocol are as follows [17, 18]. The client first creates (but does not publish) a transaction transferring the maximum anticipated total value to be payed on the channel to an "escrow" account owned by both the client and the server. Spending Bitcoin associated with this account requires the signatures of both parties.[2] The client then creates a second, refund transaction, which takes as input the escrow transaction and refunds the entire value to the client, and obtains the server's signature on it. This transaction is locked with a feature known as nLockTime, which ensures that the transaction cannot be accepted into the blockchain until a specified time window has expired. This value is set to the expected duration of the client-server relationship (a common value is one day), and allows a client to reclaim its invested Bitcoin if the server proves unresponsive or uncooperative. Finally, the client broadcasts the escrow transaction, signalling to the server that it is ready to start making payments for a service. As the server provides incremental access to a resource, the client issues a series of transactions transferring more and more of the Bitcoin in the escrow account to the server, and less and less (i.e. the remainder) to itself. These transactions bear the client's signature, but are not published; at any given point, the server could also sign one of these transactions, and claim the enclosed payment. When the client and server wish to conclude the relationship, the server simply signs and broadcasts the last transaction, which pays for the total amount of service provided. Note that this must be done before the refund transaction unlocks, or the client can empty the escrow account.

The only two transactions that are published in the channels protocol are the escrow transaction and the final payment transaction. Nevertheless, the protocol enables a client to pay a vendor for a continuous service, at as fine a level of granularity as desired, without any trust assumptions. The maximum amount that either the client or the server can abscond is the value of the last unit of service, which can be calibrated as desired [16]. This is a significant improvement over the risk involved in handing over a credit card number to an unknown party on the Internet.

2.3 The 21 Bitcoin Library

Our implementation of Bitcoin micropayments is built on a software library developed by the Andreessen Horowitz-backed Bitcoin startup 21. This library, which is coupled with a personal Bitcoin mining device and Linux machine called the 21 Bitcoin Computer, enables developers to write simple client and server applications that make and accept payments in Bitcoin. Among the packages included in the library are modules for buying and selling digital goods on a private marketplace, searching for and publishing payable endpoints (i.e. web services that accept Bitcoin), and making three types of Bitcoin transactions. These include standard Bitcoin transactions, which are

[2] Standard Bitcoin addresses are owned by only a single party. Escrow accounts, also known as multisig addresses, allow joint ownership, where the collusion of m-of-n parties is required to spend the associated Bitcoin. In this case, we utilize a 2-of-2 address to stipulate that both the client and server have to sign to transfer the deposited Bitcoin to an address owned solely by one of them.

published on the blockchain; off-blockchain transactions, which involve transferring Bitcoin addresses over the 21 network; and micropayment channels [19]. Each is packaged as a class and implements a common transaction API; this abstraction allows client-server applications to use the transactions in an interchangeable way, without any protocol-specific setup [19].

Two of the four software components involved in our prototype are Python web servers implemented with the 21 Bitcoin Library. These include a client-side module that makes payments from a user's Bitcoin wallet, and a public web server that accepts payments from clients, in exchange for granting access to its "payable" endpoints.

While we run our servers on the 21 Bitcoin Computer, a device that produces a small, steady stream of Bitcoin for programmatic use, any Linux machine with the 21 Bitcoin Library installed would suffice. The 21 Bitcoin Library itself is not a hard dependency; while it provides convenient abstractions, our software could be built with any software package that offers a Bitcoin wallet-integrated API for issuing and accepting Bitcoin transactions.

The 21 Bitcoin Library is open-sourced on GitHub. [20]

3 Related Work

Our project fits into the context of four existing services that each exhibit a particular desired property of a successful micropayments-based revenue system:

Streamium. Streamium is a service that allows users to stream live video to an audience in exchange for time-rated Bitcoin payments. The platform allows direct monetization of online courses, private lessons, gaming events and showcases, podcasts, and movies, without third-party involvement (e.g. ad networks), prior setup, or subscription contracts [21]. Streamium demonstrates an attractive use case for micropayments, namely, drawing metered payments for fluid, divisible services (e.g. live video), an idea that motivates our focus on usage-based billing schemes.

Blendle. Blendle is an ad-free portal to online journalism that was launched in the Netherlands and Germany in 2015, where it has amassed 650,000 users [22]. The service allows users to pay for content on a per-article basis, and offers users the option to refund their payment if the article does not meet their expectations. The platform cites a surprisingly low refund rate of 10%, and backing from prominent publishers, including the New York Times [22]. The budding success of Blendle signals the viability of unbundled content distribution [23], a key assumption underlying work on micropayments, and offers a roadmap for achieving widespread server-side (i.e. publisher) adoption.

Brave. Brave is a new web browser that replaces intrusive advertising and tracking devices with so-called "clean ads". The company is led by a co-founder and past CEO of Mozilla, Brandon Eich, and cites a stated goal of improving privacy and performance on the web [24]. The salient idea from Brave that we apply to our work on micropayments is the concept of opt-in advertising; in particular, the notion that since website code ultimately runs in the user's browser (and can be modified, blocked, etc.), users are entitled to choose the monetization scheme they wish to support [25]. This is reflected in the design of our prototype web service, which offers two versions of the same content – a free, ad-supported version and a payable, ad-free version.

ChangeTip. ChangeTip is a social tipping platform that enables visitors to tip writers, artists, and businesses through websites such as Twitter, YouTube, Reddit, and Facebook. To accept tips, content producers must simply publish a Bitcoin address on one or more of these outlets. ChangeTip serves as the intermediary party, allowing users to direct funds deposited to their ChangeTip accounts to the published addresses through public mentions [26]. ChangeTip's contribution to the micropayments space is its emphasis on accessibility, in that it enables even freelance publishers to directly accept money for their work from a broad online audience.

Taken together, these services offer a promising set of ideas on which to base a commercial implementation of client-server micropayments. However, we identified two key components that we found lacking in the existing body of work: namely, a fluid system of funding (users still have to pull out credit cards, or buy Bitcoin, to finance their web activity) and transactional automation (explicit action is required from the user to initiate a payment). While the use of personal mining devices, as in our prototype, may offer a solution to the first problem, in this work we focus primarily on the second issue – the problem of minimizing mental transaction costs on the payable Internet.

4 Approach

Before building our system, we identified a set of desired properties that a successful prototype should demonstrate. While we focused on resolving the cognitive load problem, we also gave critical consideration to other features that a deployable and usable system must posses, recognizing that these properties might lie on orthogonal axes, or even, opposing ones. For example, a relentless focus on automation might result in an onerous permissions model, or worse, a misleading one, deterring adoption. A browser extension that spends a user's Bitcoin in an alarming, unexpected way would not receive commendation for eradicating mental transaction costs; more likely, it would be met with a prompt uninstall and less-than-glowing reviews.

We will use our identified criteria to assess our implementation in Section 6 of this paper. Organized roughly by complexity and relevance to a research prototype, the properties are:

Fundamentals

Measurability. Can a web service accurately track and bill a client for usage?

Granularity. Can it be guaranteed that the client need never pay for more than the next unit of service, and the server need never provide service beyond the next unit of payment, for as small a unit as desired, while keeping transaction costs low?

Distribution

Ease of installation. How many different software components have to be installed by the client? How much user or system-dependent configuration is required?

Reproducibility. Is it possible to encapsulate all payments-related functionality in a library or module that web services can import into their server-side codebase?

Deployment

Security. Is the client-server channel secured to man-in-the-middle attacks? Can any sensitive data involved be protected in all cases?

Scalability. Can a web service charge, accept, and securely store payments from large number of clients simultaneously?

Error handling/recovery. Can the client and server detect and recover from a broken channel connection, or a collapse of the other node, without losing critical state information?

5 Implementation

5.1 Components

Our proposed architecture consists of four key components:

Webpage – a page or resource monetized via a Bitcoin payment requirement

Server – the backend process which handles billing and payments for the monetized service; this could ostensibly be the same server that generates the site's content, but need not be

Client payment module – a software module with access to a client's Bitcoin wallet that makes payments and generates proof-of-payment header fields on request; in our implementation, this is a private web server that runs on the client's 21 Bitcoin Computer

Client browser extension – an application installed by the user that funds her usage of payable web services, by retrieving payment header fields from the generator and appending them to outgoing requests

We implemented the monetized webpage as a static, one-page Node.js web application hosted on Heroku. The page is supported by Google AdSense advertising, which is removed if the page receives a 200 O.K. response from its payments server.

We implemented the payments server using Python and Flask. The server uses the 21 Bitcoin Library to participate in the HTTP 402 Protocol with the client, and manage received payments.

We implemented the client payment module as a private web server, also using Python/Flask. The generator makes use of the 21 Bitcoin Library to issue Bitcoin transactions to the server, and generate payment header fields.

We implemented the client browser integration as a Chrome extension in Javascript. This is the core piece of software built for this study. The extension, which is a fork of an existing Chrome extension, Requestly, uses

the chrome.webRequest API to read the HTTP headers of server responses, and append HTTP headers to a user's web requests, thereby allowing it to access payable web services.

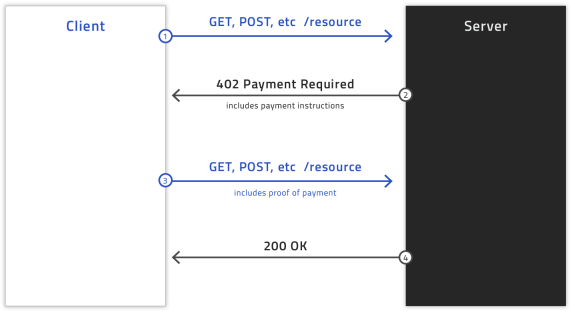
Finally, we implemented a minor modification to the 21 Bitcoin Library to support billing schemes and metered payments.

The source code for our implementation is available at https://github.com/SamvitJ/Bitcoin-micropayments.

5.2 HTTP 402 Protocol

Our implementation of micropayments builds on a core HTTP request-response protocol set forth in the 21 Bitcoin Library [19]. The protocol allows a client to demonstrate interest in a payable web service, receive payment instructions from the server, retry the request with the appropriate proof-of-payment header fields, and if the request is deemed valid, gain access to the service. These four steps are outlined in Fig. 1.

Figure 1: HTTP 402 Client-Server Protocol [19]



The HTTP code 402 corresponds to a status of "Payment Required", and is used by the server to indicate that a particular web service is associated with a fee. Alongside its 402 Payment Required response (Step 2 in Fig. 1), the server sends a set of instructional header fields, which specify the price of the endpoint and the Bitcoin address to which payments should be made (see Fig. 2).

After receiving the server's response, the client issues the actual Bitcoin transaction that pays for its usage. Though this transaction will eventually be broadcast in the blockchain, and can be verified by the server, an additional piece of information is required – an assertion (e.g. a signature) from the client that links its request for the service to the transaction that it issued.

This is accomplished by the second GET request (Step 3 in Fig. 1), in which the client sends a set of two proof-of-payment header fields – a serialized dictionary containing various metadata related to the Bitcoin transaction, such as the client's Bitcoin address and a timestamp, and an authorization field, which consists of the client's digital signature on the first header (Fig. 2). Using the client's public key, which will be included in the Bitcoin transaction, the server can verify that the same entity which issued the transaction is requesting for and receiving the corresponding service.

Figure 2: HTTP 402 Protocol Headers

Instruction Headers
Price: 100
Username: "payme.bitcoin"
Bitcoin-Payment-Channel-Server: "http://10.8.235.166:5000/payment"
Bitcoin-Address: "1BnTYEbSV1aoDRWP3wFN2fv2emprkVyMeZ"

Payment Headers
Bitcoin-Transfer: '{
"description": "http://10.8.235.166:5000/payable",
"amount": 100,
"payee_username": "payme.bitcoin",
"payee_address": "1BnTYEbSV1aoDRWP3wFN2fv2emprkVyMeZ",
"payer": "client.joe",
"timestamp": 1456779933.8278327
}',
Authorization: "KA+TuI/BP1z66a7rOF6dw5Ap7pjTkIOdmVHzQpgyA4JeDKs+ltdv4wYcQlecx/0fNkzhdYjr+FSJS0o1GsFDLQ=="

If the payment header fields are successfully validated, the server responds with the requested resource (e.g. an ad-free webpage) and an HTTP status code of 200 OK. This completes the initial request-response cycle.

The key innovation of the 402 Protocol is that it allows any entity on the Internet to *programatically* pay for usage of an online service – without having to enter into a long-standing contract (e.g. via a credit-card subscription), or defer to a third party to arbitrate (e.g. PayPal). The 21 Bitcoin Library provides an API and command line interface to this protocol, enabling developers to write standalone Python programs that represent client and server applications. These programs, however, run directly on the 21

Bitcoin Computer's operating system, and do not interact with web browsers on a user's computer or phone. In this study, we build the necessary browser integration that applies the 402 protocol to the problem of funding a user's everyday Internet activity, by listening to and modifying web requests, and interfacing with the user's Bitcoin wallet. We also extend the 402 Protocol to support continuous, metered payments, as we will discuss in Section 5.6.

5.3 Client-Side Setup

To integrate the 402 Protocol with a user's desktop web activity, we designed and built two software modules – a Chrome browser extension, to be installed on all devices from which the user wishes to make micropayments, and a headers generator, a software module that runs on the machine holding the user's Bitcoin wallet. As indicated earlier, while it is the extension's responsibility to listen for and respond to 402 Payment Required responses from web services, it is the generator that issues the Bitcoin transaction to pay for a unit of service and creates the proof-of-payment header fields that the browser uses as a token to claim the service (see Fig. 3).

In our implementation, we assumed that the device hosting the extension is *not* the same as the device running the generator, allowing our design to be generalized to a setup in which a user owns and uses multiple devices to browse the payable web (e.g. a personal computer, a tablet, a mobile phone), any one of which (or a separate device entirely) holds the user's Bitcoin and is capable of issuing Bitcoin transactions. This introduces an additional challenge - that of latency in the network communication between the browser extension and the headers generator, which while perhaps small relative to latency in the client-server connection, assuming both client devices run on the same LAN, still breaks the otherwise synchronous nature of Steps 2-6 in Figure 3.

Why does latency in the extension-generator connection require special consideration? The answer lies in the fact that the HTTP requests to the payable server (Steps 1, 7 in Figure 3) are made by the webpage on the client's browser, not by the installed extension, so any special action that the extension takes must complete in the time window between Steps 2 and Steps 7. If retrieving proof-of-payment involves a network request, as it does in our model, the webpage must 1) subscribe to a "headers received" event, issued by the extension, so that it is notified when it can retry the HTTP request to the server, and 2) implement a timeout, after which it makes the request anyway. We will discuss a potential workaround in Section 5.6.

5.4 Browser Extension

To build our browser integration, we began by forking a popular Chrome extension, Requestly, that enables users to interact with their network activity, by creating rules to, among other things, modify the headers of HTTP requests to certain domains.

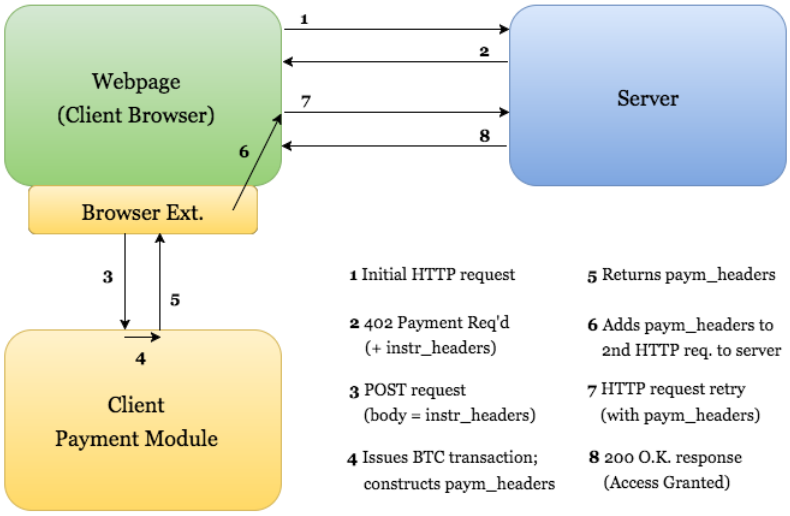
We replaced the web interface for creating static rules in Requestly with our own event-based functionality, retaining the underlying logic for reading the header fields of incoming HTTP responses, and adding header fields to outgoing HTTP requests. Both the original extension and our spin-off make heavy use of the chrome.webRequest API, which opens up a user's network activity to tampering at various points in the "life cycle" of a web request [27]. In particular, we subscribe to two key chrome.webRequest events – onHeadersReceived and onBeforeSendHeaders (see Fig. 4).

In our listener for the onHeadersReceived event, we check for the presence of the instructional header fields associated with a 402 Payment Required response (Fig. 2). If present, we store the fields in a global dictionary (Javascript object), which maps the base URL[3] of the response origin to a nested dictionary holding the instructional header fields themselves, as name-value pairs (e.g. "Price": 100, "Bitcoin-Payment-Channel-Server": "http://merchant-server.com/payment"). We also conditionally initialize an entry in a second global dictionary, which maps payable URLs to proof-of-payment header fields for the associated service.

These two global dictionaries which store instructional and payment header fields, respectively, constitute the core data structures used by our browser extension to maintain payment state. Our use of dictionaries keyed by URL allows us to support concurrent payment connections to multiple payable web services; this capability is needed, for instance, whenever a user is accessing more than one monetized domain across separate browser tabs.

[3] We use the term base URL to refer to a URL stripped of relative paths, e.g. http://www.micropayments.tech/, as opposed to http://www.micropayments.tech/js/cookies.js.

Figure 3: HTTP 402 Protocol Software Setup



After receiving a 402 Payment Required response from a domain, and updating the global dictionaries, as described above, we send off an AJAX POST request to our client payment module (Step 3 in Figure 3). The body of the request contains the instructional header fields received from the payable server; the response, captured in the success callback of the AJAX query, contains the requested proof-of-payment header fields.[4] We add the received fields, also packaged as a dictionary of name-value pairs, to the end of the payment header fields array for the corresponding URL.

In our listener for the second event to which we subscribe, onBeforeSendHeaders, we add payment header fields to outgoing HTTP requests, if they are to URLs contained in our global dictionaries. This constitutes Step 6 in Figure 3, and concludes our extension's interaction with the web request-response cycle. After using a set of header fields, we delete them from the global payment header fields dictionary, as each set corresponds to a unique Bitcoin payment transaction.

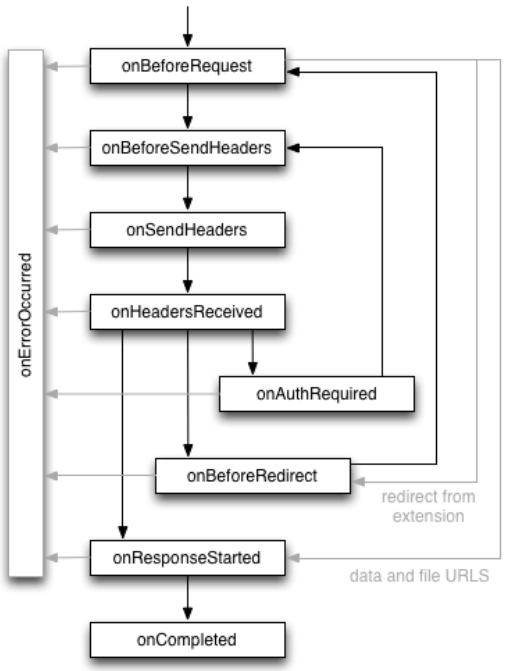
Note that in our proposed architecture, the browser extension plays merely an enabling role, providing the user with the tokens needed to gain access to gated Internet endpoints. The extension's sole responsibility is to add metadata to pre-scheduled web requests made by the monetized webpage to its backend server, activity that would occur even if the extension was not present. This separation of labor between user and vendor code enables the browser extension to pay for any kind of monetized resource (e.g. a news article, a blog, a game), given that the service implements the 402 Protocol, without special configuration or user action.

[4]The role of the client payment module is discussed in more detail in Section 5.5.

5.5 Client Payment Module

Our conception of the client payment module is comprised of a small web application built on the 21 Bitcoin Library, which responds to POST requests from the browser extension by returning a set of proof-of-payment header fields. While we use the 21 Bitcoin Library, any software package that allows programmatic invocation of a Linux-based Bitcoin wallet (e.g. a SQLite file holding private keys) would suffice. By programmatic invocation, we mean that the package must provide an API to issue Bitcoin transactions signed with a private key stored in the wallet. Note that to support the micropayment channels protocol, the library must also provide methods for opening and closing a channel, and be able to store channel-related state (e.g. the unpublished refund transaction).

Figure 4: Chrome Web Request Life Cycle [27]



5.6 Metered Payments

The infrastructure that we have described so far allows a user to visit a monetized webpage, pay the service's entrance fee (via the request-response cycle of the 402 Protocol), and gain initial access to the resource. Many websites, however, may expect a user to make metered payments for continued usage of a service. For example, a game might charge 10 cents to begin play, and 5 cents for every subsequent minute of usage, to be paid every 10 seconds. Metered billing schemes such as this one correspond well with services that involve continuous delivery of dynamic web content – like games, sites with real-time social data, such as Facebook, Twitter, and LinkedIn, which all feature dynamically generated content feeds, and audio and video streaming services, such as Spotify and YouTube.

Note that by allowing a service to specify merely two figures – an initial fee and a subsequent billing rate, we can support any linear billing scheme, from an entirely front-loaded model, such as one a stock photo provider or short-form news outlet might adopt, to a purely metered one. If a service also wishes to stipulate the frequency of payment (as opposed to accepting a standard value), a third field indicating the time window to which a billing rate corresponds is also required.

To implement support for metered payments, we amended the HTTP 402 Protocol in two ways; firstly, we added additional instructional header fields to specify a billing rate, an expiration date for the scheme, and a scheme identification number; secondly, we introduced a subsequent series of HTTP requests, to follow the initial request-response cycle, to allow a client to periodically send proofs-of-payment for continued usage of the service. That the server's initial 402 Payment Required response contains not only the value of the initial fee, but now also the billing scheme, enables the client to continue paying for the service, for as long as the scheme is valid, without any further instructions from the server. This arrangement has clear conceptual and practical advantages over one that requires the server to send instructional header fields for every payment a client makes.

On the server-side, the introduction of additional header fields requires making a minor extension to the 21 Bitcoin Library. On the client-side, the advent of persistent billing schemes, which inform future requests for payment header fields, motivates a few critical design decisions involving the browser extension. Firstly, proof-of-payment header fields are now requested both in the onHeadersReceived listener, as before, to pay the initial fee, and in the onBeforeSendHeaders listener, to pay for subsequent usage. In particular, right after a set of proof-of-payment header fields is appended to an outgoing HTTP request, the browser extension queries the generator for another set to pay for the next unit of service. Note that this payment is made in advance, so that by the time the next HTTP request is made, the headers have been received; were the generator running on the same device as the extension, it would be possible to pay synchronously. Secondly, the instructional header fields must now in fact be stored for later use, as they are accessed not only in the onHeadersReceived listener, where they inform the initial payment, but in the onBeforeSendHeaders listener as well, where the billing rate is used to fill in the price field expected by the generator. (Recall that the generator takes as input the standard instructional header fields, which specify the value of the payment transaction to be made and the Bitcoin address of the recipient, among other things.)

Finally, to cease usage of a time-rated service, a user can simply close the corresponding browser tab. Since payments are made only as long as the extension continues to query the headers generator (i.e. as long as the webpage continues to send GET requests to its payment server), closing the webpage terminates payment activity as well. This is a neat consequence of our decision to make payments via HTTP headers, as opposed to over a separate TCP connection. In our prototype, we also pause payments if the user temporarily switches out of a browser tab but does not close it, by stipulating that a GET request only be made by the webpage if the tab is currently active. While a desirable feature, note that this requires consensus between the merchant and the client on what constitutes "usage" of a resource.

5.7 Permissions Model

A key component of a successful automated micropayments system involves presenting a user with a reasonable view of their payment activity, balancing the stated goal of averting information overload with the desired properties of transparency and sensitivity to user preferences. Coupled with this issue are the trade-offs involved in developing a viable permissions model. On the one hand, one could envision a simple but inflexible system which demands up-front authorization (i.e. on install) for all actions; alternatively, one could conceive of a complex but expressive design, which takes a situational approach, but risks deluging the user with small decisions.

Our proposed solution is motivated by the observation that prompting a user to indicate a set of preferences on install is a common practice, and will presumably be tolerable, especially since the extension will be spending money on behalf of the user, but incremental configuration, where every new situation triggers a dialog box and demands user action, is less likely to be acceptable. Consequently, a well-designed permissions model should anticipate the broad categories of decisions that will arise, and accept from the user the relevant parameters on install. For instance, the browser extension could inform the user about common pricing schemes for popular payable services in various categories (e.g. news, social media, games, blogs), and then allow the user to indicate thresholds on their willingness to pay for each class of web content (e.g. at most 30 cents for a long-form journalistic piece; at most 5 cents per minute for a video). On use, the extension would check if the billing scheme for a particular website is consistent with the user's price point for that content type; if not, the extension would simply refrain from adding payment header fields to requests made by that site.[5] The user would then either be locked out of the resource or face an alternate monetization scheme (e.g. ads). Additionally, the extension could provide a preferences menu for later customization, allowing a user to calibrate their payment threshold for a particular category of web services, or specify finer-grained rules (e.g. pay up to 40 cents for Vox pieces, but at most 25 cents for all other news articles). Relegating special configuration to a menu ensures that everyday usage of the extension is free from interruption; the underlying assumption is that unless the user takes willful action, existing preferences apply.

One possible shortcoming of this approach is that before experiencing the payable Internet, the user will presumably be unable to gauge their willingness to pay for various services; thus the thresholds that the user provides on install are likely to be meaningless, and require correction after the fact. Indeed, the observation that users are more likely to deny mobile apps basic permissions (e.g. push notifications, access to contacts) when requested soon after install, as opposed to during situations that explicitly require them, suggests that users may in fact systematically underestimate their willingness to pay [28]. We note, however, that without the extension, the user would be unable to access *any* payable service; therefore, to err on the side of restriction, while allowing a user to gradually expand the set of services they choose to pay for, is arguably a more natural approach than a configuration that presupposes a user's preferences, or that explicitly surveys the user over time.

5.8 Security

Running the client payment module as a private web server entails various security risks. In particular, if a malicious party intercepts or obtains a set of payment header fields, it can claim a unit of service that was actually paid for by the owner of the generator. Why is this so? Recall that the payment header fields serve as a proof-of-payment, linking an HTTP request for a resource to the

[5] Note that this requires the browser extension be able to classify websites as belonging to a particular content type. Ostensibly, this information could be provided in yet another instructional header field.

Bitcoin transaction that paid for it (see Section 5.2). Once the transaction is made, the first party to use the associated header fields can claim the service. This attack is possible because our conception of micropayments involves no user authentication, so a server cannot link a client Bitcoin address to a particular TCP connection. Introducing user authentication would defeat some of the reasons for using Bitcoin micropayments in the first place;[6] we must look elsewhere for a solution.

Luckily, this attack can be resolved with a relatively straightforward application of standard encryption practices. Firstly, communication between the client's browser and headers generator must occur over HTTPS; this is to ensure that payment header fields are not transmitted in cleartext. This requires generating a self-signed TLS certificate on the device running the headers generator, and installing this certificate on the client's desktop computer.[7] This ensures that the client's browser will accept the generator's self-signed certificate, when the browser extension makes an AJAX POST request for payment header fields. Secondly, the client's browser must authenticate itself with the headers generator, so that the generator can verify that it is only sending payment header fields to its owner. Doing so involves exactly the opposite process. Now the user must generate a self-signed certificate on their desktop computer for their *browser*, and install it on the device hosting the generator. This second measure enables the less common practice of client authentication, needed in our case to ensure that only the user's own browser can retrieve headers from the user's generator.

Finally, another attack vector is the channel of communication between the user's browser and the payable web server. To avoid misuse of payment header fields on this channel, the client must ensure that the server is using up-to-date cryptography and owns a valid certificate issued by a reputable certificate authority.

[6] One of the major advantages of using Bitcoin micropayments over a subscription model is that it eliminates the time cost of going through a signup and login process for every site.

[7] Scripts for creating a self-signed certificate are included in the headers generator repository. We also made available our script for importing and installing the certificate on OS X.

6 Evaluation

6.1 Automation

To evaluate the degree to which our system automates the process of paying for content on the web, we begin by enumerating the set of tasks that our software performs for the user, and the set of actions, both implicit and explicit, that the user must take even with our software.

Our browser extension effectively handles three classes of problems – the logistics of making payments, the need to make "best-estimate" decisions based on previously indicated preferences, and anomaly detection and handling. In terms of logistics, the extension can inform a site that a user wishes to pay for access, pay the initial entrance fee, pay the metered charges in a timely, consistent way, and cease making payments when a user leaves the site. The extension can also made binary decisions based on a broad view of user preferences; specifically, the extension can read a site's billing scheme and decide whether or not to pay for the service (i.e. add payment header fields) based on a user's payment threshold for that content type. Of course, a user can override this decision by changing their preferences, turning off the extension, or exiting the webpage. Finally, the extension can perform basic validation of site behavior, and take action appropriately. Namely, it can check that a site's payment-related web activity (e.g. frequency of GET requests) matches up to its claimed billing scheme; if it does not match, the extension can simply refuse to pay and alert the user. In our prototype, we fully implemented payment logistics, partially implemented scheme-based decision making (we parse and store the billing scheme, but do not currently offer an interface that accepts user preferences), and did not implement anomaly detection. These three capabilities, however, are directly entailed in the system we have described thus far.

We now consider the decisions a user must make even with the extension automating some aspects of the micropayments process. To begin, the user must verify that the extension is in the desired state (i.e. on or off) before starting a browsing session. Once on the web, the user must make a mental decision before clicking on a Google search result or following a link. Since the extension will automatically pay the initial fee if the service is a payable one and the fee is below the user's indicated threshold, the

user must consider if the destination will in fact charge,[8] and if so, whether the content will be worth the initial fee in this particular instance. The user may have placed a value of 20 cents on New York Times articles, but that does not imply that the user will always be willing to pay to read one. While the extension provides a safety net, guaranteeing that the user will never pay more than some upper threshold for a resource, it is still the user's call whether to pay at all. Once on a site, a user might wish to check how much a site is charging, which requires hovering over or clicking on the extension; moreover, at any given point, the user has to decide whether it is worth spending more time on the site. Additionally, when leaving the computer, the user must deliberately switch out of or close the tab to stop making payments. Finally, the user has to perform some degree of maintenance and customization; this includes ensuring that their Bitcoin wallet is funded, that the headers generator module is running as expected, and that the extension is up-to-date with their payment preferences.

Putting together these lists, we see that our proposed system addresses the cognitive load problem primarily by automating the technical aspects of transactions, and by tying payment behavior to actions a user would normally make while browsing the web (e.g. clicking on a link, scrolling through a page, exiting a tab). This still leaves the user the responsibility of actually taking these actions, and of overriding extension behavior when necessary. While wrapping payments functionality into existing web activity reduces the burden on the user in one sense, some of the psychological weight is merely shifted over to what were previously inconsequential actions (e.g. following a link to a news article). Determining the degree to which this weight has shifted will require subsequent user studies.

6.2 Desired Properties

We now return to the desired properties identified in Section 4, and briefly evaluate our system on each criteria:

Fundamentals

Measurability. *Can a web service accurately track and bill a client for usage?* We achieve measurability by building a periodic request loop to the payment server into the Javascript of the monetized webpage. The AJAX requests in this loop fire whenever the page is active, ensuring that with every unit of resource usage, the client must issue a transaction and append payment header fields to the outgoing request. Since code that runs in the user browser can be modified by the client, usage must also be independently tracked on the server-side. This is something we did not implement in the prototype, but would be a required security feature in a commercial deployment of our system.

Granularity. *Can it be guaranteed that the client need never pay for more than the next unit of service, and the server need never provide service beyond the next unit of payment, for as small a unit as desired, while keeping transaction costs low?* Payment granularity is achieved through use of the micropayments channels protocol, discussed in Section 2.2. In particular, every metered payment corresponds to a transaction issued and signed by the client, and sent to the server, but not necessarily published. The existence of the escrow and refund transactions ensure that only the final transaction must be published; as a result, fees must only be paid on two transactions – the escrow transaction and the final transaction.

Distribution

Ease of installation. *How many different software components have to be installed by the client? How much user or system-dependent configuration is required?* The client must install the 21 Bitcoin Library and headers generator module on a machine running Linux or OS X, and the browser extension on a desktop computer with Google Chrome. The client must then purchase or transfer Bitcoin to the Bitcoin wallet associated with the generator, and configure the extension with the desired payment permissions and the IP address of the generator. Finally, the client must start the generator as a permanent server process. This is a fair amount of configuration, and likely precludes usage of the system, as it is now, by non-technical users. However, it should easily be possible to bundle the 21 Bitcoin Library and generator module in a single package, and include all the configuration steps

[8] Whether a particular service will charge or not may in fact be unclear, e.g. if the site is one the user has never visited before. However, it is reasonable to assume that with the rise of payable web services, search results snippets and HTML link tags will begin indicating initial fees, eliminating any guesswork from the process of following a link.

(wallet setup, starting the server) in a standard GUI-based install process. This reduces the setup process to the installation of just two software components – the Chrome extension and the wallet/generator package. Note that more work is required if the client wishes to implement recommended security measures, namely the use of HTTPS on the extension-generator channel (see Section 5.8). However, it is reasonable to assume that this setup can also be included in the install process for the extension and the generator.

Reproducibility. *Is it possible to encapsulate all payments-related functionality in a library or module that web services can import into their server-side codebase?* Less than 50 lines of code are required to setup a simple Flask server that offers free (ad-supported) and payable endpoints to users. This is possible through our use of the payment.required Python decorator, a feature in the 21 Bitcoin Library that performs payment header fields validation for a server route (e.g. /payable/timerated). The best way to incorporate payments functionality into a complex, existing server codebase may simply be to add the decorator to the appropriate endpoints, and thus make use of it in an application-specific way. Note that using the decorator, and storing received payments, requires installing and importing modules from the 21 Bitcoin Library.

Deployment

Security. *Is the client-server channel secured to man-in-the-middle attacks? Can any sensitive data involved be protected in all cases?* If the client and server use HTTPS to communicate, the client should be protected from man-in-the-middle attacks, and payment header fields passed on the channel should be inaccessible to eavesdroppers. To guarantee that no client-side attacks are possible, the client's browser and headers generator should also use HTTPS to communicate, and the browser should be required to authenticate itself to the generator (see Section 5.8).

Scalability. *Can a web service charge, accept, and securely store payments from large number of clients simultaneously?* This is a property that we admittedly did not test for, and that a commercially deployed system must be built to support. Note that most server-side web frameworks written for higher-level languages, such as Flask, support concurrent client connections by default, and our payments model is built on this assumption. The specific scalability bottlenecks that payments functionality might introduce involve payments state and load balancing (i.e. a client may need to pay the initial fee and subsequent metered charges to the same server replica) and security (i.e. isolating servers with wallet credentials from client-facing servers).

Error handling/recovery. *Can the client and server detect and recover from a broken channel connection, or a collapse of the other node, without losing critical state?* Our current implementation of the payments server is stateless, though for reasons discussed in the section on measurability, this will not be a true of a commercial implementation. On the client side, one possible problem could stem from the fact that, to make metered payments, the browser extension must retrieve and store payment header fields in advance. This means that if either the client or server terminate the connection before the headers can be used, and the billing scheme corresponding to the headers expires, the client would have paid for a unit of service it will not receive. This problem is not unsolvable, however; the server could credit the payment toward future purchases, or the client could simply discard the headers and accept the small associated cost. Additionally, if a connection breaks after the client has paid the initial fee (e.g. Internet connection fails, server goes down, client simply refreshes or exits the page), and the client subsequently revisits the page, it may not be reasonable, depending on the billing scheme, to charge the client the initial fee again. In our prototype, we solved this problem by introducing a browser cookie, which is created when the client pays the initial fee and deleted when the corresponding billing scheme expires. Once again, since cookies are subject to tampering, additional state may need to be introduced on the server to accurately track a client's payment history.

7 Conclusion

In this paper, we presented the architecture for an automated micropayments system that enables users to make payments to web services without taking explicit, separate action. A key component of our solution involves wrapping resource billing schemes and fee payments in the initial web activity of a page, through the use of special HTTP header fields. Moreover, the binary state design of our browser extension, which either takes or refrains from taking payment action based on its prior configuration, allows the user to assume a higher level role while navigating the payable web. We also demonstrated a natural extension of our system to support metered payments, a comprehensive permission model, and a solution to a possible security threat. We concluded by discussing the ways in which our system achieves the stated goal of automation, while still exhibiting various other desired properties, such as transactional granularity, reproducibility, and recoverability.

8 Acknowledgments

Thanks to the 21.co support team for answering our questions about the 21 Bitcoin Computer.

References

[1] T. VanToll, "The web's cruft problem," http://developer.telerik.com/featured/the-webs-cruft-problem/, July 2015.

[2] G. Aisch, W. Andrews, and J. Keller, "The cost of mobile ads on 50 news websites," http://www.nytimes.com/interactive/2015/10/01/business/cost-of-mobile-ads.html, October 2015.

[3] J. R. Mayer and J. C. Mitchell, "Third party web tracking: Policy and technology," in *IEEE Symposium on Security and Privacy*, 2012.

[4] F. Manjoo, "Fall of the banner ad: The monster that swallowed the web," http://www.nytimes.com/2014/11/06/technology/personaltech/banner-ads-the-monsters-that-swallowed-the-web.html, November 2014.

[5] P. Sawers, "Pay-by-bundle: Curing subscription fatigue," http://thenextweb.com/media/2014/03/28/subscriptions/, March 2014.

[6] G. Ferenstein, "The psychology behind the new york times paywall," http://www.fastcompany.com/1740113/psychology-behind-new-york-times-paywall, March 2011.

[7] C. Shirky, "Fame vs fortune: Micropayments and free content," http://www.shirky.com/writings/fame_vs_fortune.html, September 2003.

[8] N. Szabo, "Micropayments and mental transaction costs," http://szabo.best.vwh.net/berlinmentalmicro.pdf, March 2011.

[9] C. Shirky, "The case against micropayments," http://archive.oreilly.com/pub/a/p2p/2000/12/19/micropayments.html, December 2000.

[10] A. Odlyzko, "The case against micropayments," in *Financial Cryptography*, 2003.

[11] P. C. Fishburn, A. M. Odlyzko, and R. C. Siders, "Fixed fee versus unit pricing for information goods: competition, equilibria, and price wars," in *Internet Publishing and Beyond: The Economics of Digital*

Information and Intellectual Property, B. Kahin and H. R. Varian, Eds. Cambridge, MA: MIT Press, 2000, pp. 167–189.

[12] C. Dixon and B. Evans, "Advertising vs. micropayments in the age of ad blockers," http://a16z.com/2015/09/25/a16z-podcast-advertising-vs-micropayments-in-the-age-of-ad-blockers/, September 2015.

[13] "Paypal fees," https://www.paypal.com/webapps/mpp/paypal-fees, 2016.

[14] "Transaction fees," https://en.bitcoin.it/wiki/Transaction_fees, March 2016.

[15] "Bitcoin fees for transactions," https://bitcoinfees.21.co/, 2016.

[16] S. Yassami, N. Drego, I. Sergeev, T. Julian, D. Harding, and B. S. Srinivasan, "True micropayments with bitcoin," https://medium.com/@21/true-micropayments-with-bitcoin-e64fec23ffd8#.sft1o4nvi, February 2016.

[17] M. Carlsten, H. Kalodner, and P. Ellenbogen, "Bitcoin micropayments applied to proxy rotation," https://github.com/hkalodner/java-socks-proxy-server/tree/master/paperBitcoin, January 2015.

[18] "Working with micropayment channels," https://bitcoinj.github.io/working-with-micropayments, June 2013.

[19] T. Julian, "The 21 bitrequests library (two1.bitrequests)," https://21.co/learn/21-lib-bitrequests/, 2016.

[20] "Get the free 21 bitcoin library," https://21.co/free/, 2016.

[21] "Streamium," https://streamium.io/, 2016.

[22] "Would you pay for journalism if you could get your money back on clickbait?" http://www.theverge.com/2016/3/23/11286072/blendle-micropayments-journalism-money-back-clickbait, March 2016.

[23] A. Kopping, "Blendle: A radical experiment with micropayments in journalism," https://medium.com/on-blendle/blendle-a-radical-experiment-with-micropayments-in-journalism-365-days-later-f3b799022edc, April 2015.

[24] "Brave software | building a better web," https://www.brave.com/, 2016.

[25] "Brave's response to the naa: A better deal for publishers," https://www.brave.com/blogpost_4.html, April 2016.

[26] "More than a like, better than a share / changetip," https://www.changetip.com/, 2016.

[27] "chrome.webrequest," https://developer.chrome.com/extensions/webRequest, 2016.

[28] B. Mulligan, "The right way to ask users for ios permissions," https://techcrunch.com/2014/04/04/the-right-way-to-ask-users-for-ios-permissions/, 2016.