Bitcoin Gateway

A Peer-to-peer Bitcoin Vault and Payment Network

Omar Syed & Aamir Syed
http://arimaa.com

July 26, 2011

Abstract

We propose the introduction of a peer-to-peer network for storing bitcoin wallets and mapping email addresses to bitcoin addresses to provide a safe and easy to use experience for non-technical bitcoin users. The resulting bitcoin system will be able to provide features similar to centralized online payment systems such as PayPal while maintaining the decentralized goal of bitcoin. We feel that this key infrastructure software will be critical to facilitating wider adoption of bitcoin.

1. Introduction

The early adapters of bitcoin have been mostly technically skilled individuals who are well aware of how bitcoin works and what they need to do in order to keep their bitcoins safe. As more non-technical users acquire bitcoins they will not be computer savvy enough to manually encrypt their wallet and make a remote backup of it. Without proper precautions the new wave of users are bound to have their bitcoins stolen or lost due to their own errors. This will lead to a bad end user experience and will cause millions of potential bitcoin users to shy away from it. Online services such as banks for bitcoins can help to alleviate the problem of securely storing bitcoins, but they are counter to the decentralized nature of the bitcoin project. The low transaction fees possible with bitcoin are no good if all your savings can be lost or stolen in an instant. In the end, the average losses will cost more than the transaction fees of conventional currencies. If the bitcoin community develops a bitcoin gateway to safely store bitcoins in a peer to peer data cloud, bitcoins can be stored more safely and securely than any currency stored in a bank. In addition, this gateway can also be used to simplify the end user experience of sending and receiving bitcoins. The development of this key supporting infrastructure will be crucial to massive adoption of bitcoin in the future.

This document will try to provide a vision of what could be possible with bitcoin if a payment gateway layer is added. Some technical details are provided to show possible implementations.

2. User Experience

Ensuring that bitcoin users have a safe and smooth experience is critical to the success of bitcoin. With regard to safety, a bitcoin user should not be expected to know more about safety than what is currently required by online banking web sites. A user can expect to authenticate using the client to gain access to their bitcoins. Beyond this the user should not be expected to encrypt or backup their bitcoin data files to keep the bitcoins safe. This service needs to be seamlessly provided to the users.

Although most current bitcoin users store and access their bitcoins using a desktop client, the new wave of users will more likely use mobile device to send and receive bitcoins. It is imperative that the bitcoin client be provided not just on desktop platforms but also on mobile platforms. In fact, the client software needs to support a wide range of platforms.

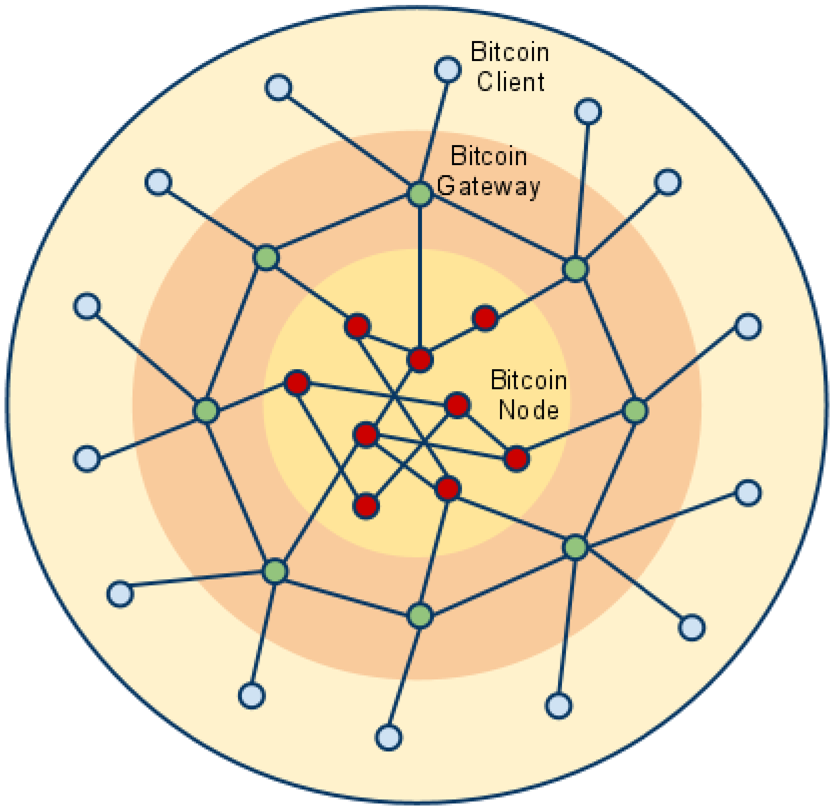
The start up time for the client also needs to be very fast even the first time it is started. Current bitcoin software has a slow start up time due to downloading the block chain needed to confirm transactions and generate bitcoins. Generating bitcoins need not be supported by end user clients since the compute resources needed for this would be much more than available on end user devices.

With regard to ease of use the process of sending bitcoins needs to be very similar to how PayPal users send money to each other. A user should be able to simply send bitcoins by specifing the recipient using an email address rather than a bitcoin address. The client should take care of looking up the bitcoin address using the provided email. It is also essential that the user be able to specify some notes with the payment, which can be viewed by the recipient along with the payment. Being able to see a note and the senders email address along with the payment would allow the recipient to more easily process the payment.

Currently, bitcoin does not provide any way for merchants to bill a user. However, this feature is needed by merchants to automate their online stores. A merchant should be able to use the email address provided by the customer to send a bill which then appears in the customer’s bitcoin client and can be accepted or rejected by the customer.

3. Architecture

There are three key software components needed to provide a safe and smooth user experience. These components include: the bitcoin node, bitcoin gateway and the bitcoin client. The bitcoin node provides the functions of the current headless bitcoin client. There are no changes needed to the current bitcoin software for this component.



The bitcoin gateway provides an interface between the bitcoin client and the bitcoin node. It communicates with the bitcoin node using the bitcoin P2P wire protocol and it communicates with the bitcoin client by providing a web API. The bitcoin gateway also communicates with other gateway servers to create a structured peer-to-peer network and provide distributed cloud storage services using a Distributed Hash Table (DHT). A Kademlia type P2P network can be used to implement the DHT (see http://en.wikipedia.org/wiki/Kademlia). This DHT can be used to store mappings of email address to public key and mappings of public key to wallets, bitcoin addresses and other data.

The bitcoin client is a web application that resides on the user’s local device. It consists mostly of HTML and JavaScript code that becomes active when opened by a standards compliant web browser. The user obtains the web application from a trusted source and saves it on the local device; similar to downloading a bitcoin client program from a trusted source. The technically savvy user can view the source of the HTML file and run a check-sum to verify it’s authenticity. The bitcoin client uses XMLHttpRequest to send messages to the bitcoin gateway. All communication between the client and gateway is over a secure https connection. The ability to connect to the bitcoin gateway is possible due to the gateway allowing open cross site access to its web APIs (see http://www.w3.org/TR/cors/ and http://www.w3.org/TR/access-control/). The client uses a local web storage to save persistent data. For example the wallet file or user preferences can also be saved by the client to local web storage (see http://www.w3.org/TR/webstorage/).

The DHT maintained by the bitcoin gateway provides a distributed global database that is accessed and modified by the bitcoin clients according to rules enforced by the gateway. The clients send messages to the gateway along with a signature which the gateway can use to validate the request. The structure of the message sent by the client is:
message = {request_nonce, signature}
signature = {encrypt2(hash(request_nonce), private_key_sender)}
request_nonce = {request, timestamp_msec, public_key_sender}

The signature is a hash of the request containing a timestamp and user’s public key all encrypted with the private key of the user operating the client. A millisecond resolution timestamp which also serves as a nonce is included with the request to prevent future reuse of the request. The request is considered invalid 15 seconds after the given timestamp.

In this document, the function hash(message) refers to a cryptographic message digest algorithm (see http://en.wikipedia.org/wiki/Cryptographic_hash_function), the function encrypt2(message, key) refers to an asymmetric key encryption algorithm (see http://en.wikipedia.org/wiki/Public-key_cryptography) and the function encrypt(message, key) refers to a symmetric key encryption algorithm (see http://en.wikipedia.org/wiki/Symmetric_key_algorithms). Recommended algorithms to use for these functions are: SHA-256 for the hash function, AES-256 for the symmetric key encryption function, and ECC-512 for the asymmetric key encryption function.

In describing the operations of the bitcoin client and bitcoin gateway this document will focus mainly on create and read operations and not dwell into update and delete operations. The intent of the description is not to provide a full specification, but rather to illustrate some key operations.

4. Bitcoin Gateway

The bitcoin gateway server is intended to run as a daemon program and stay connected to the gateway network for long periods of time. It is expected that most gateway servers will be operated by merchants that provide an online shop and accept payment in bitcoin. The bitcoin gateway can be run on the same host computer that runs the web server for the online store. Individuals who host their own web sites on a Virtual Private Server (VPS) should also consider running a bitcoin gateway to help strengthen the network.

When the gateway server is first started (from a command shell), it prompts the user for the port number to use and also to set the admin username and password. Only strong passwords are accepted. The gateway server generates a public/private key pair. It stores the port, username public key and the private key (encrypted with the password) in the configuration file. The password is not stored on disk or any other permanent storage and is only held in memory while the gateway server is running. The administrator needs to provide the password each time the server is started. When initially run the gateway server will not have a node id. Thus it does not try to connect to other peer nodes and instead waits for secure https connections. It prints the URL at which it is waiting for connections. For example https://mydomain.com:8866.

The gateway server can be configured from the relative URL /admin by providing the admin username and password. A gateway server will not be able to participate in the bitcoin gateway network until it has been approved and given a node id. To obtain approval the admin needs to fill out the application form and submits it to the bitcoin gateway network. The application form is under the /admin area of the gateway web interface. Some fields that need to be submitted include contact name, email, phone, country, ip address, port number, and public key of the gateway server. The gateway server comes preconfigured with the URL of some gateway network nodes that it can connect to. The only operation it will be allowed to do before approval is submit a “join gateway” message with it’s application form.

Upon approval the gateway server will be contacted by one of the nodes in the gateway network and given a random number that is the node identifier for this gateway server. The gateway server stores this in the configuration file. The gateway network (not server) stores:
<key, value> = <hash({identifier, “gateway”}), {public_key, ip_address, port_number}>

This allows the gateway server to connect to any node in the gateway network and prove that it has been approved by providing it’s identifier and answering a challenge to prove that it owns the public key associated with the identifier. In addition the gateway servers IP address is also checked to see if it matches that stored in the DHT.

Once connected to the gateway network the gateway server becomes a node in a Kademlia type peer-to-peer network
(see http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html).
The node uses a local database such as SQLite to provide a local store for the key value pairs. When a gateway server receives a message from the client it validates the request before executing it. If the request was valid and requires retrieving a value for the key specified in the request, the gateway performs a FIND_NODE operation to find the 20 nodes that have a node id closest to the key specified in the request and issues a FIND_VALUE operation to these nodes to obtain the value associated with the key. It sends the most occurring value to the client. If the request was valid and requires storing one or more <key, value> pairs it finds the 20 nodes that have a node id closest to the key specified in the request and passes the message to these nodes. The nodes each validate the message and modify their local storage if the message is valid. This is similar to a STORE operation, but instead of providing only the value to store the original message is provided so that the gateway servers that store the value can validate the message before deciding to store it. If the message is not valid the value is not stored.

The gateway validates requests by checking that the request hash matches the signature hash:
request_hash = hash(request_nonce)
signature_hash = encrypt2(signature, public_key)

The clients sets the value of the request_nonce and signature as described in the “Architecture” section. If request_hash and signature_hash are the same then this establishes that the request originated from the owner of the public_key and that operations in the request which effect the public key can be carried out. It also establishes that the request was not changed since being signed by the sender. The gateway should also check that the current time is not more than 15 seconds after the timestamp given in the request to prevent the message from being reused.

With each DHT entry stored in the local database the gateway server keeps a timestamp that gets updated whenever the value is modified or accessed. This timestamp can be used to find DHT entries that have not been used for a long time (like 5 years) and remove them.

The gateway server must be capable of upgrading the software automatically. The configuration file should provide the URL of some software repository and the gateway server should periodically check for updates, download them and run the new version. The administrator of the gateway server cannot be expected to keep up on installing critical updates and running a newer version.

5. Bitcoin Client

Before starting the bitcoin client users on platforms that allow multiple processes should exit all programs and run a program (such as Task Manager on Windows) to show the user what other programs are running at the same time. The user should be careful that no rouge programs are running while running the bitcoin client. The bitcoin client should remind the user to do this when it is started.

Any password or access pin entered into the client should be entered by mapping the keyboard in a random way and showing the user an image of the keyboard mapping in order to prevent a program from determining the password by logging key strokes. Although this should be the default behavior the client should allow the user to turn off random keyboard mapping if the user feels the environment on the local device is safe and does not require it.

When the bitcoin client is opened with a web browser on the users device, it initially connects to a bitcoin gateway servers from a list of servers that is hard coded into the client. The client then requests a list of other servers. The gateway server may consider the IP address of the client to suggest a list of servers that are near to it. When the client needs to send a message to the gateway it may pick from any one of the gateway servers it knows about. The user can also enter the URL of the initial gateway server to connect to. The user can also specify the only gateway server the client should connect to.

5.1 Registering an Email

In order to deal with bitcoins by using email addresses instead of a bitcoin addresses, the users must associates their email address with a public key. These associations are stored in the bitcoin gateway and effectively provide a distributed public key infrastructure with no central authority.

The process of associating an email address with a public key requires the user to perform a few steps similar to validating their email address on a web site. The user selects the "register email" link in the bitcoin client. The client generates a key pair to be associated with the email address and prompts the user for the email address and initial password (entered twice). The user is given feedback on the strength of the password and only strong passwords are accepted. The client sends a message to the gateway with:
request = {"register email", email, id_private_key, enc_private_key}
id_private_key = hash({public_key})
enc_private_key = encrypt(private_key, stronger_password)
stronger_password = hashN({password})

The hashN(input) function is just the hash(input) function but iteratively hashes the output N times where N is a number like 1,000,000. This makes the user entered password stronger before using it to encrypt the private key. Trying to crack the password by guessing requires an equivalent effort of hashing for each guess.

The gateway generates a validation number and emails it to the email address given in the request.
validation_number = hash({client_message, random_number})

The gateway stores the request as:
<key, value> = <hash({validation_number, "register email"}), client_message>

Note that the client message received by the gateway is {request_nonce, signature} as defined in the Architecture section.

The user checks the email account for the validation number that was sent by the gateway and pastes it into the "validate email" form of the client. The client sends a message to the gateway with:
request = {"validate email", validation_number}

The gateway uses the validation number to retrieve the original "register email" client message. The gateway validates the "register email" request by decrypting the signature of the "register email" message with the public key given in the "validate email" message to see if it matches the hash of the request_nonce of the "register email" message.

If the "register email" message is valid the gateway stores the public key given in that message as:
<key, value> = <hash({email, "public key"}), public_key>
stores the private key given in that message as:
<key, value> = <hash({id_private_key, "private key"}), enc_private_key>
and stores the email given in the message as:
<key, value> = <hash({public_key, "email"}), email>

However, if the "public key" entry already exists the gateway will not replace it unless the client has also provided an encrypted version of the validation_number which can be decrypted using the public key currently associated with the email. To replace the public/private key associated with the email the client would need to send a request as:
request = {"validate email", validation_number, enc_current_validation_number}
enc_current_request_hash = encrypt2(validation_number, private_key_current)

A sequence of "register email" and "validate email" requests can also be used to change the password associated with the email.

When the public key and private key are initially generated the client should store them in the local web storage of the device to prevent losing them if the client is restarted between the "register email" request and "validate email" request. The key value pairs to store are:
<key, value> = <hash({email, "public key"}), public_key>
<key, value> = <hash({id_private_key, "private key"}), enc_private_key>
<key, value> = <hash({public_key, "email"}),email>

5.2 Login

A user that has already registered can login by selecting the "login" link in the client and providing the email and password previously registered.

The client sends a message with:
request = {"get public key", email}
to the gateway to get the public key associated the email address. The client sends a message with:
request = {"get private key", id_private_key}
id_private_key = hash({public_key})
The client can now use the public key and private key as needed to make requests to the gateway on behalf of the user.

The client should be careful not to store the user's password on disk or any other permanent storage. The password should not even be saved in memory on devices where the web browser or operating system allows the memory to be accessed by another process. The client should require the user to enter the password for each operation that requires decrypting the private key to avoid saving the password in memory. Although this should be the default behavior the client should allow an option for saving the password in memory if the user feels the environment on the local device is safe and does not require entering it each time it is needed.

5.3 Wallets
A wallet file contains public/private key pairs. The public key is hashed to produce the bitcoin address and the private key is used to prove ownership of the bitcoin address. A wallet file can contain additional data associated with each key pair such as a label.

A user creates a wallet by selecting the "create wallet" link in the client and picking a name for the wallet. The user can optionally provide a pin code for accessing the wallet. The client randomly selects a password for the wallet. To store a wallet file in the DHT the client sends a message to the gateway with:
request = {"set wallet", id_wallet, enc_wallet}
id_wallet = hash({private_key, wallet_name, wallet_pin})
enc_wallet = encrypt(wallet, wallet_password)
wallet_password = the client randomly selects a wallet password

The gateway stores the wallet as:
<key, value> = <hash({id_wallet, "wallet"}), enc_wallet>

To store the wallet password in the gateway the client sends a message to the gateway with:
request = {"set wallet password", id_wallet, enc_wallet_password}
enc_wallet_password = encrypt2(wallet_password, public_key)

The gateway stores the wallet password as:
<key, value> = <hash({id_wallet, "wallet password"}), enc_wallet_password>

Note that to change the wallet name or wallet pin the client needs to save the wallet and wallet password using the new wallet id and delete the old wallet and wallet password with the old wallet id.

To retrieve the bitcoin wallet file, the user selects the "open wallet" link and enters into the bitcoin client the wallet name and access pin (if one was selected). The client sends a message with:
request = {"get wallet", id_wallet}
id_wallet = hash({private_key, wallet_name, wallet_pin})

The client also sends a message with:
request = {"get wallet password", id_wallet}
to get the password to decrypt the wallet. The client can now decrypt the wallet and access the keys it contains when needed. The user can maintain multiple independent wallets that are uniquely identified by the wallet name. When multiple wallets are open, the user can transfer bitcoin addresses from one wallet to another.

The client should store the wallet data in both the gateway and local storage. However, before saving the wallet file and wallet password to the local storage the client should prompt the user for confirmation. Although the user should create and open wallets from a trusted device, the user may be on an untrusted device and does not want to save the wallet data locally. If the wallet information does not already exist in local storage the user should be prompted before storing. If the wallet information has already been saved locally the client can update it without prompting the user.

5.4 Get Transactions
To show the user the current balance and previous transactions the client sends a list of bitcoin addresses to the gateway and gets back a list of transactions for the specified addresses. The gateway continually receives the block chain from the bitcoin network and maintains a database of transactions keyed on bitcoin addresses. When the gateway receives a "get transactions" request it can use this database to make such queries fast. For each transaction the results should include: the bitcoin address, the bitcoin value, the transaction hash, the out section index, and the transaction hash which spends these bitcoins unless it has not yet been spent.

5.5 Send Transaction
When the user wants to send bitcoins to someone the client uses the list of unspent transactions to construct the transaction to be sent to the

gateway which will forward the transaction to the bitcoin network. The client will need to ask the user for the wallet access pin to be able to open the wallet and access the private keys temporarily. The client can then sign the addresses being spent to provide proof for validating the transaction. When a wallet file is opened the wallet password and the private keys should not be saved in local storage or kept in the bitcoin client memory space after use. The bitcoin addresses, public keys and other data can be saved in memory beyond immediate use.

5.6 Send Payment by Email

The client can generate and associate bitcoin addresses with an email so that others can send payments using just the email address. The client sends a "set addresses" request to the gateway with:
request = {"set addresses", list_of_addresses}
Each entry in the list of addresses consists of
address = {bitcoin_address, enc_bitcoin_address}
enc_bitcoin_address = encrypt2(bitcoin_address, private_key)

The gateway stores this as:
<key, value> = <hash({public_key, "addresses"}), list_of_adresses>

The client saves the key pairs of bitcoin addresses that are pending receipt of payment to a wallet named "incoming payments" which the user does not have to explicitly create or open. When a payment is received for an address in this wallet the user is asked to select which user created wallet to move the payment (bitcoin address) to.

To send bitcoins using an email address, the user selects the "send to email" link in the client and is prompted to enter the email address, the amount to send and any notes to include with the payment. To initiate the send the client first needs to map the email address to a public key with:
request = {"get public key", email_recipient}
However, to avoid being fooled by a rouge gateway server the client should make the "get public key" request to several gateway servers and ensure that all return the same value.

The client can now obtain a bitcoin address by sending a message with:
request = {"get address", public_key_recipient}

The gateway retrieves the list of addresses using the given public key and picks an unused address entry to send to the client. The gateway returns:
address = {bitcoin_address, enc_bitcoin_address}

Upon receiving the entry the client verifies the bitcoin address by decrypting the encrypted version of the address it with the public key associated with the email.

If the client wants to reserve this bitcoin address for a transaction, it sends a message with:
request = {"reserve address", public_key_recipient, bitcoin_address, sender_enc_bitcoin_address, enc_note}
sender_enc_bitcoin_address = encrypt2(bitcoin_address, private_key_sender)
enc_note = encrypt2({encrypt2(note, public_key_recipient), sender_email, bitcoin_address}, private_key_sender)

The gateway can validate the request by decrypting sender_enc_bitcon_address with the public key given in the message to check if it matches the bitcoin_address given in the request. The gateway marks that this address has been taken by modifying the entry for the given address to include the signature given in the request. Thus the address entry becomes:
address = {bitcoin_address, enc_bitcoin_address, public_key_sender, sender_enc_bitcoin_address, enc_note}
The gateway will not use this address in future "get address" requests.

If the client included a note with the transaction the gateway can decrypt the encoded note with the senders public key given in the message and verify that it contains the senders email and the bitcoin address.

The client can now use the bitcoin address that it reserved to send the payment by constructing a send transaction request and passing it on to the gateway to forward to the bitcoin network.

Periodically the client which associated the bitcoin addresses with its email sends a message to the gateway with:
request = {"get addresses"}
The gateway returns the list of addresses using the public key given in the message. Some addresses may have been updated to include the sender email and notes. The client can move the sender email and any note included with a used bitcoin address to the wallet file so that when the payment is received it can be processed appropriately. The client can remove the used addresses from the list and replace them with new addresses and issue a "set addresses" request.

5.7 Send Bill by Email

The send bill feature will allow bitcoin users to place an order on a merchant web site and immediately be presented with a bill that can be viewed in their bitcoin client. If the product needs a shipping address or special instructions from the user, these can be entered by the user. Once the user has accepted the bill the merchant can become aware of it immediately. Once the bitcoins are received the merchant will be able to link the payment with the user who made the payment, thus making it easy for the merchant to process the payment.

A user (usually a merchant) can send a bill by selecting the "send bill" link in the client and entering the email address of the user to bill, the amount to bill, a note and specifying if a shipping address is required. The client first obtains the public key of the user being billed by sending the message with:
request = {"get public key", email_billed}
Note that the client should ask several gateway servers and ensure that it gets the same public key from all of them. The bill can also be generated by backend order processing software.

The client generates a new bitcoin address and sends the message with:
request = {"send bill", bitcoin_address, public_key_billled, enc_public_key_billed, enc_bill}
enc_public_key_billed = encrypt2(public_key_billed, private_key_biller)
enc_bill = encrypt2(encrypt2(bill, public_key_billed), private_key_biller)

The gateway updates the entry for:
key = hash({public_key_billed, "bills"})
The value stored for this key is a list of bills. A new entry is added to this list. The bill entry contains:
bill = {bitcoin_address, public_key_billed, public_key_biller, enc_pubic_key_billed, enc_bill}
public_key_biller = public_key given in the request_nonce of the message

The client saves the key pair associated with the bitcoin address to the "incoming payments" wallet.

Periodically the client can make a "get bills" request to the gateway to retrieve the list of outstanding bills. The message includes:
request = {"get bills"}

The gateway returns the list of bills using the public key given in the message. The client displays the bills in the list to the user and allows the user to accept or reject the bill. If the user accepts the bill the client can generate a bitcoin transaction and send it to the gateway to forward to the bitcoin network.

The users response to a bill whether accepted or rejected is sent to the gateway as:
request = {"process bill", bitcoin_address, public_key_biller, enc_public_key_billed, enc_bill_response, enc_bill}
enc_public_key_billed = encrypt2(billed_email, private_key_billed)
enc_bill_response = encrypt2(encrypt2(bill_response, public_key_biller), private_key_billed)

The gateway updates the entry for:
key = hash({public_key_biller, "processed bills"})
The value stored for this key is a list of processed bills. A new entry is added to this list. The entry contains:
entry = {bitcoin_address, public_key_billed, public_key_biller, enc_pubic_key_billed, enc_bill, enc_bill_response}
public_key_billed = public_key given in the request_nonce of the message

After the user accepts or rejects the bill the client can make a "remove bills" request to the gateway passing the email address and list of bills to be removed. Each entry in the list of bills to be removed includes the bitcoin_address and the bitcoin address encrypted with the private key associated with the users email. The gateway can validate each entry and remove the bill identified by the bitcoin address.

Due to the possibility of spam, the ability to send a bill might be limited to just one per day per user. This can be enforced by the gateway server by storing the public key, timestamp and signed bill However, this would be too restrictive for merchants. To avoid the restriction a merchant can run their own gateway server that is configured to allow much higher number of bills to be sent by specified users (identified by email or public key).

Periodically the client can make a "check for payments" request to the gateway to retrieve the list of paid bills. The message includes:
request = {"get processed bills"}

The gateway returns the list of bills that have been processed using the public key given in the message. The client displays the processed bills in the list to the merchant automatically showing the status of bitcoins received for each processed bill. The merchant can move the payment received from the "incoming payments" wallet over to another wallet. The bill and bill response can also be moved to the wallet and linked to the corresponding bitcoin address. The client can issue a command to remove the processed bills from the gateway.

5.8 Forgot Password

One of the biggest problems with having strong passwords is that they are so easy to forget. Unlike a centralized bank web site which can provide customer support to help users recover their passwords there is no one that can help with a decentralized, distributed service with no central authority. Also, if a person passes away, their family members can contact the bank and recover their assets. However, with a peer-to-peer network there is no one that can help recover the bitcoins if someone passes away. These types of problems need to be addressed for bitcoin, otherwise bitcoins will eventually be lost. Users can use password management software (see http://passwordsafe.sourceforge.net/, https://lastpass.com/, http://www.clipperz.com/) to help remember the password, but it probably won't help in a situation where someone passes away without telling anyone else their master password. The bitcoin gateway can help provide a solution to these problems.

The bitcoin client should allow the user to setup a guardian who can be given the password if the user has lost access to the account. The user should choose someone who they trust to be the guardian. The guardian can use the client to request the gateway to access the user's password. The gateway does not immediately give the password to the guardian, but instead emails a message to the user that the guardian wants access to the password. When the user logs into to the bitcoin account the user is shown the request from the guardian and can cancel the guardian's request. However if the user does not cancel the request within a week the guardian can again request to access the password and this time the gateway will give the encrypted password to the guardian. The guardian can decrypt the password using the guardian's own private key and access the users account if the user has passed away. Otherwise the guardian can give the password to the user so that the user can access the account and change the password.

To setup a guardian, the user selects the "add guardian" link in the bitcoin client and is prompted to enter the email address of the guardian and the login password for the current account. The client makes a "get public key" request to get the public key of the guardian. The request should be made to multiple gateway servers and verified that they all return the same public key. The client can now send the gateway a "set guardian" message with:
request = {"set guardian", email_recover, public_key_guardian, email_guardian, enc_password_guardian}
email_recover = email of the account making the request
email_guardian = email of the guardian account given by the user
enc_password_guardian = encrypt2(password, public_key_guardian)

The gateway saves the request by updating the "my guardians" list for the user:
<key, value> = <hash({public_key_recover, "my guardians"}), [..., {public_key_guardian, email_guardian,
enc_password_guardian, email_recover}, ….]

The gateway also updates the "guardian for" list for the guardian:
<key, value> = <hash({public_key_guardian, "guardian for"}), [..., {public_key_recover, email_recover}, ..]

The gateway sends an email to the user (email_recover) to notify that a guardian has been added to the account. The user should check to make sure this email is received in their inbox and setup filters to make sure that emails from the gateway are not considered spam.

When a guardian wants to access the password the guardian selects the "recover password" link in the bitcoin client. The client makes a "get guardian for" request and is given the list of accounts which the guardian can access. The guardian selects one account from the list. The client sends a "recover password" message with:
request = {"recover password", public_key_recover}
public_key_recover = public_key of the account to recover; found in the "guardian for" list

The gateway looks at the "my guardians" list using public_key_recover and validates that there is an entry that has a public_key_guardian that matchs the public key of the guardian (given in the request_nonce of the message). The gateway checks to see if the entry has a timestamp field. If it does not the gateway modifies the entry to add a timestamp field and sets its value to the current time. The gateway also sends an email to email_recover to notify that the guardian wants to gain access to the account and how the request can be canceled.

If the timestamp does exist the gateway checks to see if the timestamp is more than 7 days old. If not it sends an email to email_recover as it did before when the timestamp was created. If the timestamp is more than 7 days old the gateway returns password_guardian to the client. The client can decrypt this with it's private key at the request of the guardian.

If the user wants to cancel the guardian's access request, the user can choose the "recover cancel" link in the bitcoin client. The user is shown the pending access requests and can pick which one to cancel. The client sends a "recover cancel" message to the gateway with:
request = {"recover cancel", public_key_guardian}
The gateway modifies the entry in the "my guardians" list to remove the timestamp.

6. Security

6.1 Sybil Attack

One of the biggest security concerns for an open peer-to-peer network that allows anyone to join is the possibility of a Sybil attack. An attacker can have a disproportionately large number of nodes join the network and eventually gain control of the network. Fortunately, gaining control of the network does not compromise the encrypted data stored in the network. There is no known technical solution to this type of attack. The only way to limit the possibility of such an attack is by increasing the effort needed to join the network and allowing only verified nodes to join. The gateway provides an initial "join gateway" request which a new node must use to submit it's credentials. Implementing a review process that verifies the contact information brings in a human element to the security process and increases the effort needed to join the network. All applications should be approved unless they are from IP addresses known to have abused the trust in the past. The application should also be rejected if the contact information given is incorrect. This would prevent an attacker from having hundreds of nodes suddenly join the network. In addition any nodes that are found to be functioning differently than expected can be removed from the network by removing their node id from the network. Nodes can periodically issue tests to one another and report abnormal behavior to a network monitoring service. This process should help deter attackers while still allowing legitimate nodes to join the network. Although this does introduce the need of a central authority for joining the network, it does not require a central server for the operation of the network. We are not particularly fond of this work around, but see no other alternative to maintaining complete decentralization while limiting the potential of a Sybil attack.

Currently the bitcoin nodes do not require any kind of review to join the network. This is primarily due to the bitcoin nodes also serving as bitcoin clients and needing to spontaneously join and leave the network. However, a separation of the bitcoin client from the bitcoin node should allow the bitcoin nodes to also require a review process before joining the network. A new bitcoin node can submit an application to the gateway and wait to receive a node id before being allowed to join. This should also strengthen the security of the bitcoin network against a Sybil attack.

6.2 DoS Attack

A Denial of Service (DoS) attack is one where the resources of the computer are made unavailable for their intended use by saturating the computer with many useless requests. A DoS attack can occur either at the network level or at the application level. Such attacks would be very difficult to defend against and could cause an outage of service.

One way to deter DoS attacks on the gateway is to setup a public website which serves as a dashboard into the health of the peer-to-peer gateway network. Each node can periodically send a status message to the website providing information about which peers have connected to it along with various metrics such as which peers have sent invalid messages, which peers have recently sent the most messages, the CPU, and network load on the server, etc. These metrics can be compiled by the website into a real-time dashboard that highlights nodes under abnormal load or observing abnormal behaviour. Such systems are commonly used in companies to monitor mission critical applications. As the transaction volume on the bitcoin network increases it will become a high value, mission critical application and require close monitoring by the community. Such a monitoring system can help to quickly identify a DoS attack and block the source of the attack.

6.3 Hash Collision

Although the hash function can be chosen to make hash collisions extremely improbable, the thought of such a collision occurring and causing someone to lose their wallet or private key is still quite nagging. To eliminate the possibility of such a loss a backup of the value can be made whenever it is stored in the DHT.

For example when the gateway stores the wallet as:
<key, value> = <hash({id_wallet, “wallet”}), enc_wallet>

It can also store a backup as:
<key, value> = <hash({id_wallet, “wallet backup”}), enc_wallet>

This will ensure that even if the original is destroyed due to a collision a backup is still available. The probability that both the origin and backup will have a collision will be extremely small. In addition keeping a local backup on a trusted device should eliminate the possibility of any data loss due to a hash collision.

7. Implementation

The bitcoin community needs to formally define the bitcoin gateway API specification and start implementing the bitcoin gateway along with the bitcoin client. The process could take about a year before reaching production level.

A small step in this direction would be to create a gateway program that only handles a “get transactions” and “send transaction” request and develop a browser based client that manages the wallet files locally. It would not yet support sending bitcoins to email addresses and other features. However it would decouple the wallet management task from the bitcoin network servers and allow faster development of each component as well as providing some immediate security benefits to end users.

Unlike conventional open source software that gets developed only when some devoted developers have time to spare, the bitcoin software can be developed at a much faster rate by providing a financial incentive for the development. Members of the bitcoin community with large bitcoin investments have in the past setup bounties to get things done quickly. Such a model for developing the gateway and client software could be very beneficial for the community as long as the resulting code base is kept free and open for further development.

8. Conclusion

Peer to peer networks where the nodes modify their local data based on cryptographic proofs provide a new paradigm for developing decentralized, trust independent systems. Such a system is quite suitable for providing a bitcoin gateway that interfaces the bitcoin nodes to bitcoin clients and provides a safe and easy to use experience for the end users.

The features and advantages provided by the bitcoin gateway and bitcoin client proposed in this paper include:

- Clear separation of the client software from the core bitcoin software, thus allowing each to evolve faster.
- The ability to run the same client program on any platform that supports a web browser, including mobile devices.
- Automatic encryption and remote storage of the bitcoin wallets, while still allowing the ability to store the wallets securely on a local device.
- Allowing users to send bitcoins by email addresses instead of bitcoin addresses.
- Allowing merchants to bill customers using email addresses and tracking receipt of payment.
- Fast client start up time.
- No need for end user devices to download the bitcoin transaction blocks.

- Ability to access ones bitcoins from anywhere without having to carry around a wallet file.
- Allowing multiple wallets with different access codes.
- Preventing the lose of bitcoins due to forgetting passwords.
- Preventing the lose of bitcoins due to death.
- No changes required to the current bitcoin software.

The core bitcoin software by itself cannot provide all the features needed to facilitate a robust end user experience. Additional layers of supporting software needs to be developed. If the end user experience with bitcoin is safe and easy, many more people will be able to start using bitcoin. We believe that the development and deployment of the bitcoin gateway is the next step in the evolution of the bitcoin system.