The Circulex Protocol
INCOMPLETE DRAFT

T. J. M. Makarios

Revision: `8349a36`
February 5, 2019

Abstract

The circulex protocol is specified. Circulex is a system for making payments via circular exchanges through chains of trust.

This is currently an incomplete draft. Also, it's probably the case that some of the things it says (particularly about other projects, like Stellar) are no longer true.

Contents

1 Introduction

1.1 Payments through chains of trust

A project now known as Ripple Classic had the idea of facilitating payments by using chains of trust relationships [21]. For example:

Example 1. Alice wishes to buy a smartphone from Frank for 600 New Zealand dollars (NZD); Frank is willing to sell the phone for 450,000 Korean won (KRW). Alice agrees to pay her friend Bob 600 NZD whenever they next meet

in person; Bob's friend Carlos owes him 500 United States dollars (USD), so Bob forgives 400 USD of Carlos's debt; Carlos agrees to pay his friend Denise the 400 USD he's gained from Bob; Denise, in turn agrees to pay her friend Edith 450,000 KRW; finally, Edith pays her friend Frank the 450,000 KRW she got from Denise, and Frank sends the phone to Alice.

In this way, everyone in the chain pays as much as they receive, or voluntarily exchanges currency with other currency (or with a smartphone) at an acceptable rate.

One thing to note about these transactions is that they are, in general, not simply paths, but cycles. In Example 1, obligations to pay money flow from Alice to Bob to Carlos to Denise to Edith to Frank, but the cycle is completed by Frank sending a smartphone to Alice. In this document, such transactions are called *circular exchanges*.

Furthermore, it is not only the payments, but also the trust relationships that form a cycle. There is at least a one-way trust relationship between Alice and Frank; Alice trusts that if Frank receives the 450,000 KRW she is trying to send him, then he will send her the smartphone.

The direction in which the payments flow is, in this document, called the *paywise* direction; the opposite direction is called the *talkwise* direction, because, in the circulex protocol, messages confirming (and, in fact, causing) the execution of a transaction are sent in that direction.

1.2 Problems and existing solutions

Automating circular exchanges requires solving two problems: the first problem is that of discovering efficient paths through which to send payments; the second is that of coordinating agreements to actually execute such payments.

A project called Stellar solves these problems by maintaining a single authoritative ledger, which records the trust relationships and outstanding debts between entities [22]. It aims to distribute authority over the ledger among many independent entities [14], but at present its network remains under the control of one organization [13]. Further, even if it successfully devolves power away from its present central authority, it is still somewhat centralized, around a global, public ledger.

However, a fully decentralized protocol implementing circular exchanges would face its own problems. Without a single authoritative ledger, chains of payments like that in Example 1 would require everyone in the chain to agree to simultaneously update their ledgers; otherwise, if Alice pays Bob and then finds out that Denise, who Alice neither knows nor trusts, has failed to pay

Edith, and, as a consequence, Frank has not sent Alice the phone, then Alice will be justifiably upset.

This is an example of the so-called *Byzantine generals problem* [15], in which a number of participants try to ensure that all of those behaving correctly agree on a course of action, even if a small number of the participants behave arbitrarily, either maliciously or accidentally.

A theorem known as the *FLP impossibility result* places a limit on what can be achieved by solutions to the Byzantine generals problem in an asynchronous setting — that is, when no assumptions can be made about the speed at which participants in a protocol perform their necessary computations, or about the speed at which messages are delivered [8]. In particular, it states that no asynchronous consensus protocol in which there is more than one possible outcome, but which prevents well behaved participants from disagreeing on the outcome, can guarantee that *any* of its well behaved participants will eventually commit to one of the possible outcomes, even if there is only one misbehaving participant, and even if its misbehaviour is simply an eventual failure to continue communicating, rather than the sending of erroneous messages.

In practice, consensus protocols avoid the FLP impossibility result either by placing requirements on the speed of computation and delivery of messages, or by ensuring that although non-termination is possible, its probability approaches 0 as time approaches infinity [10]. Such protocols can be designed to prevent disagreements between well behaved participants even if a certain proportion of the participants are acting maliciously [17].

However, in the case of circular exchanges, no participant can guarantee even a minimum proportion of trustworthy participants, since each participant in a given transaction is only assumed to have a trust relationship with the person they are paying and with the person who is paying them, and there may be arbitrarily many other participants.

1.3 Overview of circulex's solution

1.3.1 Decomposing circular exchanges

Circulex solves this problem by decomposing circular exchanges into bilateral agreements in which one participant agrees to pay the other in exchange for the other participant providing the first with proof (before a specified deadline) that the overall circular exchange is being executed.

So in Example 1, Carlos would agree to pay Denise (his paywise neighbour) on the condition that she provides him with timely proof that the whole circular exchange is going ahead. This proof (called a *complete agree-*

ment) is valuable to Carlos because he can forward it to Bob (his talkwise neighbour), thus triggering Bob's obligation to forgive 400 USD of Carlos's debt.

Because the transaction has been decomposed into bilateral agreements between participants with explicit trust relationships, any failure to properly execute the protocol has a direct negative effect only on the participant to blame for the failure, or on a participant who explicitly chose to trust the one to blame. This ensures that the incentive to prevent failures — by taking care to implement the protocol correctly and by choosing carefully who to trust — is as squarely as possible in the hands of those most able to prevent those failures.

1.3.2 Avoiding the FLP impossibility result

Circulex avoids the FLP result by requiring the complete agreement to be supplied before a specific deadline in order to trigger payment obligations; thus the protocol is not asynchronous. However, each deadline is agreed on by the two participants it affects — the one providing the complete agreement and the one receiving it. Furthermore, each participant can wait till they know the deadlines required by their potential neighbours in one direction before choosing the deadlines for their potential neighbours in the other direction. This means that they can avoid being forced to relay messages faster than they are sure is possible.

For circular exchanges, this means of avoiding the FLP result is preferable to the technique of ensuring almost certain eventual consensus while allowing for the possibility of so-called *perpetual pre-emption*. This is because many participants will not want transactions to be in limbo for an indefinite length of time, unable to rely either on their execution or on their non-execution; such a possibility would be particularly undesirable if the arbitrarily high proportion of untrusted participants in a transaction could deliberately lengthen the time in which the transaction's outcome is uncertain.

1.3.3 Speed, safety, and liveness

In order to mitigate the effects of computers unexpectedly failing or being taken offline, some systems adopt decentralized consensus protocols that ensure (in all but the most dire circumstances) safety and liveness at the cost of some speed. While circulex does not prevent participants from using decentralized consensus protocols for their own decision-making, it does not require it, either.

Instead, circulex requires each participant to nominate a number of *relays*

(including, if desired, their primary instance) to which their neighbours must send relevant complete agreements. A complete agreement is considered to have been received when more than half of the relays have received it. This ensures safety, provided that less than half of a participant's relays (and less than half of the relays of each of its neighbours) are offline or misbehaving during the time in which the outcome of an attempted transaction is uncertain.

As for speed, a participant's decision-making need not be encumbered by ensuring consensus among their network of relays; when the primary instance has received the relevant information from sufficiently many of its relays, it can immediately act on that knowledge, without even having to inform its relays of the outcome.

However, the liveness of a participant's instance is more fragile; when a participant's primary instance is offline, that participant cannot take part in any new transactions; but in a large, well-connected circulex network, the temporary absence of a single participant will have negligible effect on the liveness of the network as a whole.

One additional benefit of this system is that a participant's relays need not be trusted to the same extent as their primary instance is trusted; relays need not know the cryptographic secrets that the primary instance requires in order to create the messages it sends. However, relays must be trusted to faithfully forward the messages they receive, to accurately report the times at which they received them, and to maintain confidentiality regarding the identities of the participant's neighbours' relays.

1.3.4 Pathfinding

In order to execute a circulex transaction, the would-be participants must first find the path through their trust relationships along which the transaction will flow. This is achieved in a decentralized way as part of the process of constructing the complete agreement.

If someone receives a pathfinding request (in the form of a *partial agreement*) from someone with whom they have a trust relationship, it will specify the bilateral agreement that those two participants will execute if it is included as part of the overall transaction.

With the details of that proposed bilateral agreement — including the payment to be made to the paywise neighbour and the deadline for the complete agreement to reach the talkwise neighbour —, the one receiving the request can extend the partial agreement by choosing a potential neighbour in the other direction and an appropriate payment and deadline, thus constructing another bilateral agreement that might be part of the overall

transaction. In fact, they might construct several different such bilateral agreements, and send them to several different potential neighbours, hoping that one of their proposed bilateral agreements will be included in the overall transaction.

It might be objected that there is potential for this pathfinding algorithm to suffer from complexity that is exponential in the number of participants in the final path. While this may be true, it is also true that the computational power devoted to pathfinding can increase just as quickly; each potential participant in the transaction needs only to consider and construct those parts of pathfinding messages that affect themself and their potential neighbours.

The initial pathfinding request can be accompanied by hints for other potential participants, specifying, for example, which currency the pathfinding should aim for. This facility is intended to assist in increasing the speed and reducing the complexity of pathfinding, as well as reducing the cost of the final path.

Any hopeful participant who stands to gain from being part of the transaction (for example, from exchange rate spreads, or by charging their talkwise neighbour fractionally more than they promise to pay their paywise neighbour) has an incentive to help find (and therefore be part of) the cheapest path. They might attempt to do so by, for example, keeping statistics on which of their potential neighbours have most often been successful in completing transactions of different kinds.

1.3.5 Uniqueness and reversibility

During pathfinding for any given circular exchange, a participant might receive partial agreements from many potential neighbours in either the paywise or the talkwise direction; they might choose a number of these partial agreements and use each as the basis for one or more extended partial agreements for each of several potential neighbours in the other direction. But the participant might not have the confidence to commit to so many paywise partial agreements unless they are assured that at most one of them will be included in the final transaction; otherwise they might end up selling more of one currency than they want to, for example.

To provide such assurance, the circulex protocol includes another deadline in each bilateral agreement, besides the deadline for the paywise neighbour to prove to the talkwise neighbour that the transaction is going ahead. This second deadline, called the *reversal deadline*, is the deadline for the talkwise neighbour to prove to the paywise neighbour that the transaction is being reversed. If the paywise neighbour has proven that the transaction is being executed, then the payment becomes irreversible after the reversal deadline

has passed, unless the talkwise neighbour has, by that time, proven that the transaction is being reversed.

The form of this proof is simply proof that another path was also chosen for the same transaction; therefore, if a participant discovers — after paying one paywise neighbour for a particular transaction — that another paywise neighbour is also expecting payment for the same transaction, then they can use the complete agreement obtained from the first paywise neighbour to prove to the second that another path was also chosen for the same transaction, thus nullifying their obligation to pay the second paywise neighbour; the second paywise neighbour can then use the same complete agreement to reverse their obligation to pay their own paywise neighbour, and so on. (If the relevant reversal deadline has not passed, then the payment to the first paywise neighbour can also be reversed, by sending to that paywise neighbour the complete agreement obtained from the second paywise neighbour.)

In practice, it ought to be rare for two different complete agreements to exist for the same transaction, with each specifying a nontrivial path (that is, a path including participants other than the initiator of the transaction). This is because there's no useful reason for more than one such complete agreement to be constructed. However, it's important that this mechanism exists, in order to give participants assurance that they need not be obliged to pay more than one paywise neighbour once for each transaction.

Furthermore, the mechanism can also be used to construct transactions that are — with the consent of every participant — deliberately reversible (at the option of the initiator) for a significant length of time after they are executed. In order to reverse such a transaction, the initiator can construct a complete agreement in which they are the only participant, and send it to their paywise neighbour (before their reversal deadline) as proof that the transaction is being reversed.

The initiator might find it difficult (or even impossible) to find paths whose participants are unanimously willing to agree to a long-term reversible transaction, or they might find it more expensive to use such paths, but the protocol allows them to try.

1.3.6 Privacy

Simply by being decentralized, circulex allows more privacy than a system like Stellar that relies on a global public ledger of balances and trust relationships. However, a naive implementation of the solution described above would still reveal a great deal of information to untrusted participants in transactions (and even, during pathfinding, to hopeful participants and computers set up to act as hopeful participants for the purpose of surveillance or

interference).

To enhance privacy, the circulex protocol is designed to prevent participants in a transaction from learning either the identity of participants (other than those with whom they have a trust relationship) or the details of bilateral agreements (other than their own).

In any particular transaction (or attempted transaction), each participant is publicly identified by a temporary public key that is, in general, unique not only to that circular exchange, but also to each proposed bialteral agreement that that participant receives. This temporary public key is generated by the neighbour who proposes the bilateral agreement; they modify the participant's medium-term public key using the details of the proposed bilateral agreement, together with a shared secret obtained from the two participants' medium-term keys. Only the participant receiving the proposal (whose medium-term public key was the basis for the temporary one) is able to produce signatures that will pass verification with the temporary public key; and even if a participant's medium-term public key is known to adversaries, the connection between that public key and the temporary one cannot be established.

This achieves several things: it ensures that when a participant needs a new temporary public key for a potential neighbour, they require only one-way communication to establish it; it prevents the accidental reuse of temporary public keys; and it allows the temporary public key to uniquely specify (only to the relevant participants) both the identity of its owner and the details of the bilateral agreement.

1.4 Flexibility of trust relationships

It is worth noting that circulex does not require the trust relationships to be bidirectional. In Example 1, Bob already trusts Carlos to repay the 500 USD debt; Carlos need not trust Bob to pay 400 USD, because if there is a dispute about whether he forwarded the complete agreement to Bob on time, then Carlos can simply refuse to repay more than 100 USD. And Carlos does not need to trust Denise, either, because if there is a dispute about whether he received the complete agreement on time, then he can simply refuse to pay her the 400 USD.

In fact, in some circumstances, the trust relationships need not even be direct. Consider:

Example 2. Richie and Sally each run their own business. Each business takes deposits from customers and allows customers to cheaply transfer money to other customers of the business, as well as to other people around

the world via circulex.

Richie and Sally have no direct trust relationship, but Mark trusts both Richie and Sally to pay their debts to him, and to faithfully implement the circulex protocol.

Richie owes Mark 100 pounds sterling (GBP), and Sally owes him 400 Swiss francs (CHF). Mark instructs Sally that he is willing to forgive 300 CHF of her debt, provided that Richie increases Mark's balance by 200 GBP.

Using the circulex protocol, Richie and Sally take part in a circular exchange in which Sally is the paywise neighbour of Richie. (Mark does not implement circulex at all.) They agree that if Sally provides Richie (on time) with the complete agreement, then Richie will increase Mark's balance by 200 GBP (which Richie will recover from his talkwise neighbour in the transaction). This will allow Sally to decrease Mark's balance by 300 CHF (which she will pay to her paywise neighbour in the transaction).

If, in this example, a failure occurs in which Richie believes he has not received the complete agreement from Sally on time, but Sally believes that she did send it in time, Sally can decrease Mark's CHF balance, and Richie can refuse to increase his GBP balance. In this way, Mark — who was the person who explicitly extended trust to whichever person really was to blame for the failure — is the one who suffers the loss; Richie and Sally can implement the circulex protocol with each other without risking that the other will cause them direct loss.

If, on the other hand, either Richie or Sally has confidence that the other will faithfully implement circulex, then there are all sorts of agreements they could choose to make with each other, or guarantees they could make to their customers.

1.5 Usage models

While circulex allows a great deal of decentralization, it doesn't force users of the protocol to adopt decentralization in practice. There are a number of different usage models that participants might adopt, with varying degrees of decentralization. These models can coexist in a single circulex network, and even a single participant can adopt aspects of different models for use with different peers.

1.5.1 The hierarchical model

One possible model for using circulex is the *hierarchical model*, which is very similar to the existing banking network. In this model, each currency is

issued by a *central bank*, which is trusted by a number of *trading banks*, each of which has its own *customers*. The trading banks can use circulex to send payments to each others' customers, via the banks' relationships with the central bank. For foreign exchange transactions, the banks can use something like the existing correspondent accounts that are used for the same purpose.

As Jan Michelfeit pointed out (see [18], page 15), the advantage of the hierarchical model is that pathfinding is trivial. Even in the case of foreign exchange transactions, the path from one currency to another can be chosen in advance of any particular attempted transaction.

However, the hierarchical model's centralization creates a risk of inefficiency and rent-seeking by banks at bottlenecks of the network, such as well-connected correspondent banks.

1.5.2 The public exchange model

Like the hierarchical model, the *public exchange model* involves a number of *banks* implementing circulex in order to serve their *customers*. The difference is that in the hierarchical model, the banks record only their debts to and from their own customers, but in the public exchange model, the banks also keep track of amounts they owe to the customers of other banks, and amounts their customers are owed by other banks.

In the public exchange model, banks also allow customers to place *orders* in an *order book*, expressing their willingness to buy or sell the obligations of other banks, often denominated in different currencies. These offers can be used in the execution of circulex transactions, as in Example 2.

Because the public exchange model can use customers' orders to form links in circular exchanges, inter-bank transfers are no longer as centralized as they are in the hierarchical model. However, in practice, the public exchange model might still feature a great degree of centralization, in that, for each currency, a very few banks — or even just one — might serve most of the people who use that currency. Such centralization might threaten customers' privacy, as well as limiting their options regarding the kinds of debt obligations they can take part in.

1.5.3 The hosted IOUs model

Instead of banks and customers, the *hosted IOUs model* has a number of *hosts*, each of which has its own *members*. Unlike the previous two models, which only record debt obligations involving banks, this model features (possibly quite informal) debt obligations between one member of a host and

other members of that and other hosts; such informal obligations are often known as *IOUs*. A member of a host can instruct that host about whose IOUs they are willing to accept, up to what credit limits, in which currencies, whether they are willing to exchange some IOUs for others, and the relative value they place on those IOUs.

This model exhibits the kind of financial decentralization that the hierarchical and public exchange models conspicuously lack, allowing members a great deal of flexibility regarding who they trust and what kinds of agreements they make. However, there is still the possibility that a circulex network built around the hosted IOUs model might be dominated by a few popular hosts, which could use their market dominance in ways that undermine the public interest, including by limiting privacy.

1.5.4 The personal server model

In the *personal server model*, individuals run their own circulex instances, which record their debts to and from their chosen peers, executing and facilitating circulex transactions according to the instructions given by the owner of the instance.

This is the most decentralized model for the use of circulex, and it carries all of the benefits of decentralization, such as privacy, robustness in the event of the failure of some participants, and the avoidance of expensive bottlenecks and attempted censorship. On the other hand, it gives individuals all of the responsibility for running and maintaining secure circulex instances, including the selection of a suitable set of relays.

However, a well designed mobile application, for example, could make this very easy to do for its users, encouraging (or even requiring) them to choose a large set of relays from among those whose IOUs they trust, on the principle that if you can trust someone to pay their debts to you, then you can probably also trust them to faithfully relay messages for you. Participants who want circulex's flexibility more than a mobile application's ease of use could run more configurable circulex software on desktop computers or dedicated servers.

Finally, it is worth noting that merely having many people run circulex instances on their own equipment will not, on its own, realize the full benefits of the personal server model; it is still possible that people's trust relationships will resemble the hierarchical model, and that the network will therefore inherit many of that model's downsides. To gain all of the benefits of the personal server model, people will need to select a number of other people whose IOUs they trust, and instruct their servers about those trust relationships.

2 Typical operation

Before going on to the fine details, it's worth getting an overview of the typical operation of the protocol — how people interact with circulex applications, which messages the applications send, in what order, and what they mean.

In the circulex protocol, this can be roughly divided into three phases: establishing relationships, pathfinding for a transaction, and executing the transaction. It's worth looking at each phase from two points of view: first, the interactions involving people; and second, the messages that circulex applications send each other.

2.1 Establishing relationships

2.1.1 Human interactions

When Alice first introduced Bob to circulex, she recommended an app, which he downloaded to his phone. When Bob first opened the app, he told it the name he wanted it to use to identify him to his friends.

Alice got her computer to display her *participant identity* as a two-dimensional barcode, which Bob scanned with his phone's camera. Bob confirmed to the app that the particpant identity belonged to Alice, as it purported to do. Likewise, Bob got his app to display his participant identity to Alice's webcam, so that Alice's circulex application knew Bob's identity.

Alice then told her application that she trusted Bob for up to 3000 NZD. Bob's app asked him if he was willing to accept this extension of credit, prompting him to discuss with Alice such questions as:

- How will his debts to her be settled?
- How will they decide when to settle them?
- Who will cover any transaction costs incurred in settling them?

The circulex protocol doesn't dictate the answers to these questions; instead, it gives participants the flexibility to decide for themselves the most suitable answers, taking into account their personal relationships and circumstances; but it's very important that both participants in a trust relationship have the same understanding about the answers to those questions.

Bob accepted Alice's extension of credit, and chose to extend the same amount of credit to Alice.

Bob also exchanged participant identities with Carlos, but because they were on different continents, they did so using their favourite secure messaging app, instead of with two-dimensional barcodes. Because Bob mostly

owns and uses NZD, while Carlos mostly owns and uses USD, they agreed that Carlos's debts to Bob will be denominated in USD and Bob's debts to Carlos will be denominated in NZD. Bob chose to trust Carlos for up to 800 USD, and Carlos chose to trust Bob for up to 1200 NZD.

2.1.2 Computer interactions

While the above was happening, the applications automatically sent a number of messages, establishing relationships among each other and their relays.

Alice's participant identity included Alice's IP address and circulex port, but Bob was connecting to the internet via network address translation, so his app didn't yet know its own public IP address and port. Bob's app initiated a connection to Alice's computer, which verified the authenticity of the connection, thus learning Bob's public IP address and circulex port. Bob's app then sent a *location request* to Alice's computer, requesting that it share that information via a *location report*, which it did.

Bob's app then exchanged *invitations* with Alice's and Carlos's applications; these carried information about their current medium-term circulex public keys, their requested bandwidth limits (if any), and their lists of chosen relays. Bob's app also exchanged *statements* with his peers' applications; these messages conveyed the credit limits set by their owners, and any other requested limits on the sizes of transactions.

Because Bob had only just started using circulex, his app listed itself as its only relay. At least three relays are required in order to participate in transactions, so Alice's computer sent a *relay offer* to Bob's app, offering to act as one of Bob's relays for a certain length of time. Bob's app assumed that because Bob trusts Alice with a substantial sum of money, he also trusts her computer to act as a relay, so it added Alice's computer to its internal list of relays, in order to include it in future invitations. Likewise, Carlos's application also sent a relay offer to Bob's app, which was accepted.

Bob's app sent a *null relay request* to Alice's and Bob's applications (in their role as his relays); this requested information about the communication latency from those relays to his peers' relays (as listed in those peers' invitations). Bob's relays (including Bob's own app) estimated this latency by sending *pings* to his peers' relays and receiving *pongs* in response. Alice's and Carlos's applications then responded to the null relay requests by each sending a *latency report* to Bob's app.

These relationships are maintained by sending fresh invitations, statements, null relay requests, pings, pongs, and latency reports, as necessary. Occasionally, if two circulex instances that need to communicate are unable to do so, more location requests, location reports, and relay offers might be

required, or *relay rejections*, which an instance can use to inform its peer that the instance's relays can't relaibly communicate with one or more of the peer's relays.

2.2 Pathfinding

2.2.1 Human interactions

Alice visits Frank's website and orders a smartphone from him. His site is set up to allow payment via circulex, so Alice chooses that option. The site generates a link encoding a *payment request*, which Alice clicks, opening it in her circulex application. The payment request is for 450,000 KRW, and includes a reference, which is used to associate the payment with Alice's order.

Alice's application knows that she uses NZD, rather than KRW, so it suggests that Alice might be willing to pay up to 610 NZD in order to get the 450,000 KRW to Frank. Alice checks the payment details and confirms her willingness to pay that much. Then the application attempts to find a path for the payment.

Separately, without any involvement from Alice or Frank, other people, such as Bob and Denise, have already told their circulex applications that they're willing to exchange certain currencies at certain rates. For example: Bob — who Carlos already owes 500 USD from a previous transaction — is willing to exchange part or all of that debt for NZD at a rate of 3 NZD for every 2 USD; Denise wants to buy up to 1000 USD at a rate of 1 USD for every 1125 KRW.

2.2.2 Computer interactions

Transaction identifier When Alice confirms her willingness to make the transaction, her application first creates a *transaction identifier* to uniquely identify the transaction. The transaction identifier includes the *completion* and *finality deadlines* for the transaction. The completion deadline is the time at which the application wants to make a final choice of the path for the transaction; the finality deadline puts a limit on how long the transaction will be reversible for.

Alice's application chooses these deadlines sufficiently far in the future to allow a reasonable length of time for pathfinding, but not so far in the future that Alice will get frustrated waiting for confirmation that her payment has gone through, or so far in the future that potential participants will be reluctant to set aside funds for long enough to participate in the transaction.

Hints Alice's application then creates two *hints*: a talkwise hint to send to Frank's circulex instance (her intended talkwise partner in the transaction), and a paywise hint to send to her potential paywise neighbours.

The talkwise hint indicates that the expected value of the payment is approximately 605 NZD (the application's estimate of the actual value in NZD). This allows Frank's instance, as well as any others who receive the hint, to focus on looking for paths through potential talkwise neighbours that have successfully participated in similar NZD transactions in the past. It also allows them to rule out potential neighbours whose credit limits would prevent them from participating in a transaction of that magnitude.

Frank's instance forwards the talkwise hint to instances that it's willing to consider as potential talkwise neighbours in the transaction; each other recipient of the talkwise hint does likewise, if they're willing to participate in the transaction themselves.

The paywise hint specifies the expected value of the transaction as 450,000 KRW, and performs a pathfinding function similar to that of the talkwise hint, but in the other direction.

Because Frank is a well-known electronics vendor, Alice's application also includes an indication in the paywise hint that the payment is going to Frank; this further assists potential participants in their decisions about whose instances are likely to be able to help complete the payment — those that have participated in payments to Frank in the past are judged more likely to be able to help again now.

Alice's application could have included a similar indication in the talkwise hint, specifying that the talkwise pathfinding should aim for Alice, but this feature was disabled by default in Alice's application, since its use could have a negative effect on Alice's privacy.

Relay requests and latency reports As soon as an instance that's willing to participate receives a hint for Alice's transaction, it can send *relay requests* to its relays. A relay request asks the recipient to act as the sender's relay for a specific transaction, relaying to specified destinations the complete agreements the relay receives between the completion and finality deadlines, and reporting on how many distinct such complete agreements it received.

For example, when Bob's app recieves a hint from Alice's application, it sends a relay request to Alice's and Carlos's instances (in their roles as Bob's relays), asking them to act as relays for this transaction, and specifying Alice's and Carlos's relays as the destinations to forward complete agreements to.

Carlos's instance uses fresh pings, if necessary, to update its information

about the communication latencies to its own and Alice's relays. Then it responds to Bob's relay request with a latency report. This response indicates its agreement to act as Bob's relay for this transaction, and provides Bob's app with the latency information it will need. Alice's application does likewise.

Partial agreements Finally, Alice's application generates some *partial agreements*, which each begin a chain of firm commitments that, if they come back to Alice, could be chosen as the path for the transaction. Alice's application generates one partial agreement to send talkwise to Frank, and another for each of her potential paywise partners.

The most informative part of a partial agreement is the *bilateral agreement* that it contains. The bilateral agreement specifies the payment to be made, and also includes a *deadline* and a *reversal deadline*.

Alice's talkwise partial agreement specifies that Frank's "payment" to Alice will be an acknowledgement that he's received 450,000 KRW in payment for her order. If, before the deadline, Alice's relays prove to Frank's that a path built using that partial agreement was chosen, then Frank will be obliged to acknowledge receipt of that payment. However, if, before the reversal deadline, Frank's relays prove to Alice's that another path was chosen, then his obligation will be nullified.

Because Alice's application will itself be choosing the final path at the completion deadline, it chooses the deadline in this partial agreement to allow enough time for the proof to traverse the slowest path from Alice's application, via one of her relays, to one of Frank's relays, thus ensuring that every one of Frank's relays will receive the proof from every one of Alice's relays before the deadline, unless unexpected network problems occur; even in such a case, multiple problems would need to occur in order for Alice's application to believe that it had supplied the proof on time, and Frank's instance to believe that it hadn't.

Frank's instance then uses the partial agreement it receives from Alice to construct another partial agreement for each of his potential talkwise partners, including Edith. The partial agreement to send to Edith specifies a payment to Frank of 450,000 KRW. It also includes a deadline sufficiently later than the deadline in the partial agreement from Alice that any one of Frank's relays can forward a proof to every one of Edith's relays before the later deadline, as long as it receives it before the earlier deadline. Similarly, Frank's instance chooses a reversal deadline early enough that if, before that reversal deadline, any one of his relays receives proof that another path was chosen, then it can forward it to every one of Alice's relays before the reversal

deadline specified in the partial agreement Frank's instance received from Alice's application.

In successive talkwise partial agreements like these, deadlines become later and reversal deadlines become earlier. Reversal deadlines always have to be later than deadlines, but earlier than the finality deadline. Therefore, Alice's application chose the reversal deadline to be equal to the finality deadline, and both to be sufficiently later than the completion deadline to allow paths of moderate latency. However, a reversal or finality deadline too far in the future would risk discouraging participation by instances that dislike lengthy uncertainty about whether or not a transaction will be reversed.

At the same time, paywise partial agreements for the same transaction are also circulating. Alice's application sends one to each of the instances it's willing to consider as paywise neighbours for this transaction, including Bob. The partial agreement sent to Bob's app specifies a payment of 605 NZD and a deadline and a reversal deadline about half way between the transaction's completion and finality deadlines.

Bob's app then builds on this partial agreement to create a partial agreement to send to Carlos's instance. This partial agreement specifies that Bob will forgive 403.34 USD of Carlos's debt. It has a deadline shortly before, and a reversal deadline shortly after, the corresponding deadlines in the bilateral agreement with Alice; this allows enough time for the relevant proofs, if received on time by at least one of Bob's relays, to be forwarded on time to all of Alice's or Carlos's relays, respectively.

Carlos's instance then builds on this partial agreement to construct partial agreements for each of his potential paywise neighbours, and so on.

Denise's instance receives two talkwise partial agreements for this transaction, but at first it doesn't consider Carlos as a very likely talkwise partner for the transaction, so it doesn't send a partial agreement to his instance. However, when it receives a paywise partial agreement from Carlos's instance, it knows that there's a paywise chain of willing participants from the transaction's initiator (who's anonymous to Denise) to Carlos, so it sends a talkwise partial agreement to Carlos's instance. It bases this on the partial agreement it received from Edith, because the other one it received specifies a larger payment for Denise to make.

It also uses the partial agreement it received from Carlos to create a paywise partial agreement to send to Edith.

2.3 Execution

2.3.1 Human interactions

No further human intervention is required to execute the transaction.

Once it's been executed, Alice's application notifies her that the payment has been made, and that it cost her only 600 NZD — a little less than the maximum she'd authorized. Frank's website marks Alice's order as paid, initiating the process to deliver the smartphone to her.

Other participants in the transaction's chosen path aren't notified about that transaction specifically, but next time they look, they'll see a change in their balances of obligations to and from their neighbours; their total net balances of obligations won't have changed, except for participants who explicity chose to allow their total balances to change (for example, by trading currencies, like Bob and Denise, or by forgiving doubtful debts at a discount, which no-one in this example did).

Potential participants who weren't included in the chosen path remain completely oblivious to the transaction's existence, unless they inspect their instance's logs, and even in that case, they can't tell whether the transaction was successful, or whether its initiator simply committed to the transaction's non-execution.

2.3.2 Computer interactions

By the completion deadline, Alice's application has received a number of partial agreements from her potential neighbours in the transaction. For example, it received a paywise partial agreement from Frank's instance, specifying that Alice would pay her paywise neighbour 605 NZD, and a talkwise partial agreement from Bob's app, specifying that she would pay Bob 600 NZD.

The partial agreement from Bob's app specifies the smallest payment for Alice to make, so her application uses that one to construct a complete agreement. It sends the complete agreement to each of Alice's relays, which forward it to the relays of all of Alice's potential neighbours, which in turn forward it to the relays of all of the potential neighbours of those potential neighbours, and so on.

For example, Alice's application, acting as one of its own relays, forwards the complete agreement to (among others) Bob's and Carlos's instances, in their roles as Bob's relays; in its role as another of Bob's relays, it forwards the complete agreement to all of Carlos's relays. For another example, Bob's app, as one of its own relays, forwards the complete agreement to all of Alice's and Carlos's relays.

In this way, the relays of all of the potential participants receive a copy of the complete agreement. There's a significant level of redundancy involved in this process, to ensure that only widespread message loss or delay can prevent the majority of any potential participant's relays from receiving the complete agreement in a timely way.

When each relay first receives the complete agreement, it sends to its primary instance (the instance for which it's acting as a relay) a *receipt*. The receipt contains a copy of the body of the complete agreement and notes the time at which the relay first received it.

For example, Bob's app receives receipts from Alice's and Carlos's instances. These allow it to conclude that the bilateral agreements with Alice and Carlos have been triggered by the timely receipt of a relevant complete agreement; however, as far as Bob's app is aware, there's still, at this point, the possibility that the obligations arising from those bilateral agreements will be nullified by the timely receipt of a different complete agreement for the same transaction.

When sufficiently many receipts reach a potential participant not included on the chosen path, it can immediately rely on the fact that it hasn't been included.

At the finality deadline, each relay sends to its primary instance a *tally* indicating the number of distinct complete agreements it received for this transaction — in this case 1 (or 0, for any relay that lost its network connection at the crucial moment).

When Bob's app receives tallies from Alice's and Carlos's instances, it can conclude that no second complete agreement reversed the effect of the first, and that therefore Alice now owes Bob 600 NZD more, and Carlos owes Bob 400 USD less. These balance changes (and an indication of which transaction caused them) are reflected in the next statements Bob's app sends to Alice's and Carlos's instances; when Bob's app receives corresponding statements from Alice's and Carlos's instances, it can verify that there's no disagreement between Bob and his peers about what their balances are or which transaction caused the changes.

2.4 Missing messages

Certain messages that contain important information — such as statements — are usually sent only once each, without the redundancy associated with complete agreements. If such a message fails to reach its destination, the intended recipient can, when it realizes that one or more messages have gone missing, send a *missing information request*, which requests that the messages be resent. The recipient of the request can respond by simply resending the

messages or by sending *quoted messages* containing the missing messages; the choice between the two depends on the type of information requested and whether resending unquoted messages might lead to a misunderstanding.

For example, Bob's app receives a statement from Carlos's instance covering a period after Alice's payment to Frank, but it hasn't yet received a statement that includes that transaction. Without any human intervention, Bob's app sends a missing information request to Carlos's instance requesting statements covering the relevant period; Carlos's instance responds by resending the statement that Bob's app hadn't received.

3 Prerequisites and other standards

3.1 RFC 2119

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [3].

3.2 Communication with peers

A circulex instance is assumed to have a number of *peers* — other instances with which it is mutually willing to attempt to execute circulex transactions.

Instances MUST be able to maintain secure, authenticated, timely communication with their peers. In order to achieve this, it is RECOMMENDED that instances communicate using DTLS [20] over SCTP [23]. Instances MUST also employ a number of relays and SHOULD take into account the number of relays a peer is using when deciding whether to make new bilateral agreements with that peer. Instances MUST keep their peers informed of changes to the set of their own currently live relays.

The default SCTP port for circulex messages SHALL be 25443.[1]

Sometimes an instance needs to instruct its peer or its relay regarding how to securely communicate with its own or its peers' relays, respectively. If DTLS is to be used, then the instance can use for this purpose subjectPublicKeyInfo objects, as defined in RFC 5280 [5], encoded according to the Distinguished Encoding Rules (DER) referenced in that document.

[1] Internet Best Current Practice 165 [6] prohibits the use on the public internet of assignable ports (0–49151) until they have been assigned to a specific service. Therefore, until SCTP port 25443 has been assigned to circulex, instances MUST use the ports reserved for experimental use (1021 and 1022) or the Dynamic Ports (49152–65535).

3.3 JSON

The circulex protocol requires the use of JSON texts [4] for specifying the terms of bilateral agreements. Such texts MUST NOT be or contain (directly or indirectly) any JSON object that has multiple members with the same name. Also, such texts MUST NOT be excessively artificially enlarged by the addition of large or numerous irrelevant components. All JSON strings MUST represent strings composed entirely of Unicode characters [24].

3.4 Encodings

In circulex messages, JSON texts and other strings SHALL be encoded according to UTF-8 [25].

Some JSON strings are used to communicate arbitrary data. In such cases, the data is encoded in base64 [12], and the resulting string, omitting padding characters, is used as the contents of the JSON string.

Except where otherwise specified, numbers included directly in circulex messages (that is, not within a JSON text) SHALL be encoded as 32-bit unsigned integers with big-endian byte order.

3.5 Signature scheme

Using the notation of *EdDSA for more curves* [2], circulex's signature scheme is based on PureEdDSA with the following parameters: $q = 2^{255} - 19$; $b = 256$; the 255-bit encoding of $\mathbf{F}_{2^{255}-19}$ is the usual little-endian encoding of $\{0, 1, \ldots, 2^{255} - 20\}$; H is SHAKE256 with 512 bits of output [7], and when its output is to be used as an integer, it is to be interpreted as the little-endian encoding of an integer in $\{0, 1, \ldots, 2^{512} - 1\}$; $c = 3$; $n = 254$; $a = -1$; $d = -\frac{121665}{121666}$; B is the point $(\ldots 202, \frac{4}{5}) \in E$; and $\ell = 2^{252} + 27742317777372353535851937790883648493$. (These parameters are the same as those of Ed25519-SHA-512, as defined in *EdDSA for more curves* [2], with the exception of the choice of H.)

Define functions h_i for $0 \leq i < 512$ such that for each x we have $H(x) = (h_0(x), h_1(x), \ldots, h_{511}(x))$; let $s(x) = 2^{254} + \sum_{3 \leq i < 254} 2^i h_i(x)$, so that $\underline{A} = \underline{s(k)B}$ is the public key corresponding to the secret key k.

When an instance wants to generate a public key for itself (either a medium-term one or a temporary one), it generates a new secret key k and proceeds exactly as in PureEdDSA with the above parameters. When it generates k, it MUST choose it randomly (or pseudo-randomly) using a uniform distribution over a set of at least 2^{256} options. Choices of different secret keys

MUST be probabilistically independent of each other. An instance SHOULD NOT reuse a public key when it's unnecessary to do so.

An instance can also, whenever it wants to, generate a public key for one of its peers, such that only that peer is able to create signatures that are valid under that public key. For this purpose, each instance is REQUIRED to have — for each peer it is willing to have as a neighbour in a transaction — a medium-term secret key; the corresponding public key MUST be known to the relevant peer. It is RECOMMENDED that these medium-term keys are regularly changed, to minimize the impact of any unwitting disclosure of the secret keys.

Suppose instance 0 has medium-term key pair $(\underline{A_0}, k_0)$, and instance 1 has medium-term key pair $(\underline{A_1}, k_1)$; each instance's medium-term public key is known to the other instance. Instance i calculates a point $A_{i,1-i} \in E$ as follows: $A_{i,1-i} = s(k_i)A_{1-i}$. It follows that $A_{1,0} = A_{0,1}$ because $s(k_1)A_0 = s(k_1)s(k_0)B = s(k_0)s(k_1)B = s(k_0)A_1$; this is a shared secret point in E for instances 0 and 1.

Each instance MUST verify that the shared secret point is not $(0, 1)$; if one of the instances finds that it is $(0, 1)$, this indicates that the other instance has chosen a public key by a non-standard method and that anyone who knows its public key can easily forge its signatures and deduce its shared secret point with compliant instances.

Suppose instance 1 wishes to provide instance 0 with a temporary public key $\underline{T}$ in a way that enables instance 0 (and only instance 0) to construct signatures that pass PureEdDSA verification under $\underline{T}$. Instance 1 generates some data x and sends it to instance 0. Each instance calculates $T = H(\underline{A_{0,1}}, x)A_0$; only instance 0 knows a discrete logarithm of T base B, specifically $t = H(\underline{A_{0,1}}, x)s(k_0)$.

Given a message M on which instance 0 wishes to construct a signature that passes verification under the public key $\underline{T}$, it calculates $r = H(h_{256}(k_0), \ldots, h_{511}(k_0), x, M)$. (Notice the inclusion of x in the input of H; without it, instance 1 might be able to induce instance 0 to sign the same message multiple times with the same value of r, but with different values of t, leading to the discovery of $s(k_0)$ by instance 1.) Instance 0 then calculates $R = rB$ and $S = (r + H(\underline{R}, \underline{T}, M)t) \bmod \ell$ and outputs the signature $(\underline{R}, \underline{S})$.

This signature will pass ordinary PureEdDSA verification on the message M under the public key $\underline{T}$, but the choice of r means that it is not *the* PureEdDSA signature of M under any particular key k. Even so, the warnings ([2], page 3) regarding different choices of r have been heeded; this signature scheme ensures that r is still chosen deterministically, is no more likely to be repeated than in ordinary PureEdDSA, and is no more guessable

than in that scheme.

Because this signature scheme uses the same verification procedure as PureEdDSA, it is just as secure against forgeries of signatures as PureEdDSA is, assuming all the necessary secrets remain secret.

3.6 Timekeeping

Instances MUST keep their clocks accurate, so that they share with their peers a common understanding of the deadlines specified in the messages they exchange. It is RECOMMENDED that instances use a standard protocol like NTP [19] for this purpose.

In this document *time label* means a TAI64N label [1]. When this document specifies that a time label is to be included in a message sent between instances, that time label MUST be encoded in external TAI64N format.

Because many systems handle leap seconds poorly, instances SHOULD be aware of leap seconds and, when in their proximity, act on conservative assumptions about the accuracy of their own and their peers' clocks.

3.7 Host addresses

In this document, *host address* means a string containing just one of the following:

- a fully-qualified domain name [16],
- an IPv4 address in dotted decimal notation [16], or
- a text representation of an IPv6 address, as specified in section 2.2 of RFC 4291 [11].

4 Definitions

4.1 Instance location

An *instance location* is a JSON object that can be used to specify a way of connecting to a particular instance, that the sender believes is valid at the time it's sent. The particular instance the instance location relates to is determined by the context that the instance location is included in.

An instance location has a `hostAddress` member whose value is a host address, specifying an address at which the instance can be contacted, for the time being.

An instance location can also have either of the following members, which are respectively REQUIRED if a non-default trasport-layer protocol or port is to be specified:

`transportProtocol`
: A JSON text specifying the transport-layer protocol to be used when connecting to the instance. If this member is omitted, its value is assumed to be `"SCTP"`.

`port`
: A JSON text specifying a valid port of the specified transport-layer protocol. When SCTP is being used, this MUST be a JSON number, if it's included. If this member is omitted, its value is assumed to be `25443`.

4.2 Instance identity

An *instance identity* is a JSON object that can be used to specify how an instance will authenticate itself to its peers. It can also include information about how to connect to the instance, but this information might change over time if the instance has no domain name, and a dynamic IP address.

If an instance uses a non-default encryption protocol with a peer, then the instance identity it shares with that peer MUST include an `encryptionProtocol` member, whose value MUST be mutually understood by the instance and its peer; additionally, in such a case, the instance and its peer MUST share an understanding of the interpretation of the other members of the instance identity, in that context. If the `encryptionProtocol` member is omitted, its value is assumed to be `"DTLS"`.

An instance identity MUST include the following members:

`name`
: A JSON text indicating how the instance will be identified during authentication. When DTLS is being used, this MUST match the common name or an alternative name of the end entity in the certificate presented by the instance.

`trustAnchor`
: A JSON text specifying a trust anchor to be used by the recipient of this instance identity for authenticating connections made with the specified instance. When DTLS is being used, this MUST be a JSON string containing just the base64 encoding (omitting padding characters) of the first 512 bits of output of the SHAKE256 hash of the DER encoding

of a subjectPublicKeyInfo object that MUST appear in the certificate chain that the specified instance presents in order to authenticate itself. This is analogous to DANE-TA certificate usage with SPKI selector and a custom matching type [9].

`locations`
: A JSON array of instance locations, specifying ways the sender believes the specified instance can be contacted, for the time being. This array can be empty, if such information is unavailable to the sender, or if it's irrelevant in the context of the message the sender is composing.

4.3 Participant identity

A *participant identity* is a JSON object that specifies a member of an instance. It has the following members:

`instance`
: An instance identity.

`username`
: A JSON string specifiying the username of a member of the specified instance. It's RECOMMENDED that the owner of the instance reserves the username "`root`" for their own use, but actually using that username is OPTIONAL.

4.4 Currency

A *currency* is a JSON object with the following members:

`unit`
: The unit of account of this currency. It is RECOMMENDED that this value is a string containing just the widely used capitalized code representing the currency, where that would be unambiguous; for example, COP for Colombian pesos, or BTC for bitcoins.

`denominator`
: A JSON number representing a positive integer. Quantities in this currency are specified using another integer, which represents the numerator of a fraction whose denominator is the value of this member; the value of the fraction is the amount of this currency that is being specified in that situation.

4.5 Payment request

A *payment request* is a JSON object that's used to specify a request for a payment to be made via circulex. It has the following members:

`payee`
: A pariticipant identity, specifying the circulex participant to be paid, according to the request.

`currency`
: The currency in which the payment is requested.

`numerator`
: A JSON number representing a positive integer, encoding the numerator of the fraction that specifies the amount requested.

It's RECOMMENDED that a payment request also includes a `reference` member, specifying what the payment is for. If the requested payment is actually made, this reference SHOULD be included in the `reference` member of a payment specification that forms part of the bilateral agreement between the payer's and the payee's instances.

4.6 Circulex version information

The *circulex version information* consists of the following eight bytes:

bytes 0 **to** 3
: The string "`cclx`" (`0x63636c78` in hexadecimal).

bytes 4 **to** 7
: The version number 1 (`0x00000001`).

Version number 0 is reserved for experimental implementations of possible future versions of the circulex protocol; it MUST NOT be used except between instances that share an understanding of the nature of the experiment.

Implementations of circulex version 1 MAY accept messages whose version information specifies a different version number, provided that those messages comply in all other ways with circulex version 1.

4.7 Bandwidth limit

A *bandwidth limit* is a member of the set $\{0, 1, \ldots, 2^{32} - 2\}$ encoded as a 32-bit unsigned integer with big-endian byte order. The use of the number

$2^{32} - 1$ in the place of a bandwidth limit is reserved for future extensions to this protocol.

When included in a message sent between circulex peers, a bandwidth limit is interpreted as the sender's desired upper limit on the short-term average bandwidth used by a certain class of circulex messages, measured in bytes per second; the particular class that it relates to is determined by context. The limit excludes the overheads associated with the relevant transport and encryption protocols.

The recipient of a bandwidth limit isn't obliged to make precise calculations of the bandwidth used by the relevant class of messages; an estimate is sufficient. If the estimates are significantly in error, the sender of the bandwidth limit can compensate by specifying an adjusted bandwidth limit in future messages.

4.8 Transaction identifier

A *transaction identifier* consists of the following 48 bytes:

bytes 0 **to** 3
: A number determining the length of a valid complete agreement for the transaction, and thereby specifying the maximum number of bilateral agreements that can be included in the transaction.

bytes 4 **to** 15
: A time label specifying the *completion deadline* of the transaction — that is, the time at which the initiator wishes to construct a complete agreement for this transaction. An instance that might have already received a complete agreement for a transaction after its completion deadline MUST NOT make any new commitments regarding that transaction that it consequently might not be able to keep. An easy way to satisfy this prohibition is simply to refrain from sending any partial agreements or latency reports for a transaction after its completion deadline.

bytes 16 **to** 47
: A temporary public key belonging to the initiator of the transaction. Instances SHOULD NOT use the same public key for more than one transaction.

Partial agreements and complete agreements are considered to belong to the same transaction if and only if they have identical transaction identifiers.

4.9 Account specifier

An *account specifier* is a JSON object that is included in certain messages sent between instances. It specifies a member of the sender's instance, a member of the recipient's instance, and a currency, and is used to establish the context for interpreting other data, such as information about obligations between those members, or changes to those obligations. It has the following members:

`myUser`
: The username of a member of the sender's instance.

`yourUser`
: The username of a member of the recipient's instance.

`currency`
: A currency.

4.10 Range

A *range* is a JSON object representing a range of possible values. It can have a `minimum` member, a `maximum` member, both members, or neither member. The value of each member that is included is a JSON number. If the `minimum` member is omitted, then the range has no lower limit (or the lower limit is the lowest meaningful number, such as 0, if negative numbers would be meaningless in that context); if the `maximum` member is omitted, the range has no upper limit (or the upper limit is the highest meaningful number in that context).

The units are determined by context; for example, the context might include a currency object, in which case the meaning of the range can be specified so that the values of the members are the numerators of fractions specifying amounts in the indicated currency, with the denominator indicated by the value of the `denominator` member of the currency object.

4.11 Balance summary

A *balance summary* is a JSON object used as part of a message communicating the sender's beliefs about the balances of obligations between a member of the sender's instance and a member of the recipient's instance. It also communicates the sender's intentions regarding future transactions that might affect that balance of obligations. It has the following members:

`account`
: An account specifier. The remaining members of the balance summary, described below, specify amounts or ranges in the currency specified in the `currency` member of this account specifier. In each case, a quantity in the currency is specified by a fraction in which the denominator is the value of the `denominator` member of the currency object; the numerator is an integer represented by a JSON number used either directly as the value of the member (in the case of an amount) or as the value of its `minimum` or `maximum` member (in the case of a range). In the case of balances, a positive number represents an obligation owed by the specified member of the sender's instance to the specified member of the recipient's instance; a negative number represents an obligation in the opposite direction.

`openingBalance`
: A JSON number indicating the outstanding obligation at the start of the period specified by the context in which this balance summary is included, taking into account all relevant bilateral agreements whose reversal deadlines are strictly earlier than the start of that period.

`closingBalance`
: A JSON number indicating the outstanding obligation at the end of the period specified by the context in which this balance summary is included, taking into account all relevant bilateral agreements whose reversal deadlines are strictly earlier than the end of that period.

`balanceLimits`
: A range indicating the current intentions (when the message is sent) of the specified member of the sender's instance regarding the balance of obligations in the near future. This range SHOULD include a `minimum` member, indicating that the sender is unlikely to participate in transactions that might cause the balance of obligations to drop below that value; similarly, if a `maximum` member is included, then it indicates that the sender is unlikely to allow the balance of obligations to rise above that value.

`transactionSizes`
: A range indicating the sender's intentions (if any) regarding the absolute amount by which it will allow any one transaction to alter the balance of obligations. Because this range relates to an absolute amount, negative values of its members are not meaningful.

A balance summary MAY also include a `maximumOutstanding` member, whose value is a JSON number indicating the sender's intentions regarding the maximum absolute amount by which it will allow the transactions of uncertain disposition at any one time to alter the balance of obligations.

The `balanceLimits`, `transactionSizes`, and `maximumOutstanding` members do not alter the balance of obligations at any point in time, or affect the effectiveness of any past, current, or future bilateral agreements involving the specified account; they are merely indicative of which bilateral agreements the sender is likely to agree to on behalf of its member.

4.12 Target

A *target* is a JSON object with some combination of the following members, none of which are compulsory:

`host`
: An instance identity.

`username`
: This member is only meaningful if the `host` member is also included. The value of this member indicates the username of a member of the specified host.

`currency`
: A currency. It is RECOMMENDED that the value of this member is specified in a form that will be widely understood. In particular, if the `host` member is also included, then the value of this member SHOULD be in a form that will be understood if it is included in a payment specification sent to the specified host. The currency it specifies SHOULD be one used by the specified host, and, if the `username` member is also included, then it SHOULD be a currency used by that member of the host.

`approximateValue`
: This member is only meaningful if the `currency` member is also included. The value of this member is a JSON number encoding a positive integer, specifying the numerator of a fraction whose denominator is the value of the `denominator` member of the `currency` member, indicating the approximate value of an intended transaction, as measured in the specified currency.

4.13 Latency probability object

A *latency probability object* is a JSON object with the following members:

`latency`
: A JSON number representing an integer — to be interpreted as a number of nanoseconds.

`probability`
: A JSON number between 0 and 1.

Such an object, when included in a message sent by a relay to a primary instance, is to be interpreted as an assertion that the relay can, at least until a particular expiry time, recognize a complete agreement for a particular transaction and ensure it reaches a particular destination within at most the specified number of nanoseconds, with at least the specified probability. The particular expiry time, transaction, and destination are determined by the context of the message in which the latency probability object is included.

The value of the `latency` member specifies the one-way delay from the relay to the destination, but includes the time taken for the relay to process the incoming message, determine that it is a valid complete agreement for the transaction, and send it to the destination. The relevant measurement is from the time (according to the relay's clock) at which the relay has finished receiving the complete agreement, until the time (according to the destination's clock) at which the destination has finished receiving it.

Because the start and end times are measured by different clocks, it is theoretically possible that the value of the `latency` member might be negative. However, implementations are strongly advised to be cautious in such situations; clock corrections might suddenly and significantly alter the measured latency.

Furthermore, in such a situation, the destination might legitimately ignore a complete agreement that it believes it has received before the completion deadline, but which the sender believes it has sent after that deadline. Therefore, if a primary instance receives a message containing a latency probability object with a negative value of the `latency` member, then it SHOULD NOT rely on the contents of that message — or on the behaviour of that relay or destination, until the instance is sure that each has an accurate clock.

The relay that sent the report of negative latency SHOULD attempt to reconcile its clock with that of the destination, aiming for the correction of whichever clock is in error. The owner of an instance with more influence over the relevant destination than the relay has might be able to hasten the correction (if the problem is with the destination) by reporting the problem to the owner of the destination.

4.14 Latency profile

A *latency profile* is a JSON object with the following members:

`destination`
: An instance identity.

`profile`
: A JSON array of latency probability objects that relate to the destination specified in the `destination` member. The context in which the latency profile is included determines the relevant expiry time and transaction for the latency probability objects.

The sender of a latency profile SHOULD aim to be as informative as is reasonably practical, taking into account the information that will be most useful to the recipient. This implies that each included latency probability object will have the lowest value of the `latency` member that the sender is willing to report for the given value of the `probability` member. However, in cases of doubt, the sender SHOULD make somewhat conservative assumptions.

In the absence of a specific understanding regarding the information most desired by the recipient, it is reasonable to assume that the recipient will be more interested in information regarding latencies that are achievable with high probability, and will not want very fine-grained information about latencies achievable only with lower probability.

The `profile` member SHOULD NOT include any elements whose information is logically implied by the inclusion of any other elements. For example, if one element states that latency of no more than 120 ms is achievable with a probability of 0.9, then another element that states that latency of no more than 121 ms is achievable with a probability of 0.8 is redundant, and its exclusion shortens the message to be sent, which is advantageous.

4.15 Signed payment path

A *signed payment path* of length m consists of the following $96m + 52$ bytes:

bytes 0 **to** 3
: Either the string "`tkws`" (`0x746b7773`) or the string "`pyws`" (`0x70797773`). The former indicates that this path is listed in talkwise order, so that each successive participant is the talkwise partner of the previous one; the latter indicates that the path is listed in paywise order.

bytes 4 **to** 51
: A transaction identifier in which the number specified the first 4 bytes is at least m.

bytes $32j + 20$ **to** $32j + 51$ **(for each** j **with** $1 \leq j \leq m$**)**
: A temporary public key. The owner of this public key is either the owner of the previous public key (who is the initiator when $j = 1$), or the talkwise or paywise neighbour of the owner of that key, depending on what was specified in the first 4 bytes of this signed payment path.

bytes $64j + 32m + 52$ **to** $64j + 32m + 115$ **(for each** j **with** $0 \leq j \leq m - 1$**)**
: A valid signature — under the key specified between bytes $32j + 20$ and $32j + 51$ — of the first $32j + 84$ bytes of this signed payment path (that is, until the end of the key after the key under which this signature is valid).

4.16 Payment specification

A *payment specification* is a JSON object used in a message sent between instances to specify the details of a proposed payment between members of those instances. A payment specification has at least the following members:

`paymentType`
: Either the string `"record"` or the string `"acknowledge"`. The value `"record"` indicates that the payment is to be made by recording an increase in the amount the talkwise partner owes the paywise partner, a decrease in the amount the paywise partner owes the talkwise partner, or both, if the paywise partner initially owes the talkwise partner an amount smaller than the total payment to be made. The value `"acknowledge"` indicates that within the protocol, the "payment" to be made by the talkwise partner to the paywise partner is simply the acknowledgement that the talkwise partner has (perhaps indirectly) received from the paywise partner an amount equivalent to the specified amount. This can be useful if such an acknowledgement triggers the talkwise partner's obligation to supply goods or services to the paywise partner, arising from an agreement made outside the protocol; it can also be useful if the paywise partner wants to make a donation to the talkwise partner, or if the paywise and talkwise partners are actually both memberships held by the same person on different hosts.

`account`
: An account specifier, specifying the currency of the payment and the partners in the payment. The context in which the payment specification is used determines which instance is the talkwise instance and which is the paywise instance, and therefore also determines which partner is the talkwise partner, and which is the paywise partner.

`numerator`
: A JSON number that is an integer representing the numerator of the fraction specifying the amount of the specified currency to be paid.

A payment specification MAY also have a `reference` member. This can be useful, for example, when the value of the `paymentType` member is `"acknowledge"`, to convey information about what the payment is for.

Example 3. In the scenario of Example 1, Alice wants Frank to actually send her a smartphone, not merely record the fact that he owes her a smartphone. So Alice orders a smartphone from Frank, who agrees to send her one if she pays him 450,000 KRW via circulex. He gives her an order reference, "441a25a", which he asks her to include in her payment, so that he can match her payment with her order. Alice's instance could send to Frank's instance the following payment specification as part of a message attempting to initiate a circulex payment:

```
{
  "paymentType": "acknowledge",
  "account": {
    "myUser": "Alice",
    "yourUser": "Frank",
    "currency": {
      "unit": "KRW",
      "denominator": 1
    }
  },
  "numerator": 450000,
  "reference": "441a25a"
}
```

If this payment specification is used (in a context in which Alice's instance is the paywise peer of Frank's instance) as part of the basis of a successful circulex transaction, then Frank will have acknowledged having received the equivalent of 450,000 KRW from Alice; this will trigger his obligation to send her the smartphone.

Example 4. Gareth is a gardener who pays for goods and services via circulex by promising to do gardening work for his friends and relatives. He maintains his own circulex instance, which he uses to record (among other things) the number of hours of gardening work he owes each of his friends; his friend Hermione is a member of a circulex host run by someone else. This payment specification is for 1 hour and 20 minutes of work to be done in Hermione's garden (assuming Gareth's instance, as the talkwise peer, includes it in a message to Hermione's circulex host):

```
{
  "paymentType": "record",
  "account": {
    "myUser": "root",
    "yourUser": "Hermione",
    "currency": {
      "unit": "Hours of Gareth's labour",
      "denominator": 60
    }
  },
  "numerator": 80
}
```

4.17 Bilateral agreement

A *bilateral agreement* is used as part of a message sent between two instances to establish potential deadlines, public keys, lists of relays, and payment details for a transaction. The context in which it is sent determines which instance is the talkwise instance and which is the paywise instance.

A bilateral agreement consists of the concatenation of the following data in the following order:

bytes 0 **to** 11
: A time label representing this bilateral agreement's deadline.

bytes 12 **to** 75
: The first 512 bits of output of the SHAKE256 hash of an invitation message previously sent by the talkwise instance to the paywise instance. This establishes the medium-term public key and list of relays employed by the talkwise instance for the purposes of this bilateral agreement.

bytes 76 **to** 87
: A time label representing the reversal deadline for this bilateral agreement. This reversal deadline MUST be later than the deadline specified in bytes 0 to 11.

bytes 88 **to** 151
: The first 512 bits of output of the SHAKE256 hash of an invitation message previously sent by the paywise instance to the talkwise instance. This establishes the medium-term public key and list of relays employed by the paywise instance for the purposes of this bilateral agreement.

remaining bytes

The UTF-8 encoding of a JSON array of payment specifications, which specify one or more payments to be made by one or more talkwise partners (who are members of the talkwise instance) for the benefit of one or more paywise partners (who are members of the paywise instance). Multiple payment specifications in the array specify components of a composite payment; they are *not* options from which one of the partners or instances may choose one. The final "`]`" closing this JSON array MUST NOT be followed by any whitespace characters.

The instances MUST execute the specified payments if both of the following are true:

- before this bilateral agreement's deadline, a complete agreement built using this bilateral agreement has reached more than half of the talkwise instance's relays; and
- more than half of the paywise instance's relays have received no other complete agreement for the same transaction before the reversal deadline (not even a different complete agreement that was also built using this bilateral agreement).

If a relay is also the initiator of a transaction, and it creates a complete agreement for it, then it is considered to have received that agreement at the first point in time at which it begins sending it to any other instance or relay.

So that relays are not obliged to keep records of all complete agreements that they receive relating to transactions they do not yet know anything about, a relay MAY ignore the fact that it has received, before the completion deadline of the transaction it relates to, any complete agreement that it receives at such a time. Also, so that relays are not obliged to accept connections and carefully examine messages received from arbitrary sources, for the purposes of a bilateral agreement, a relay is REQUIRED to accept connections and complete agreements from the relays of the other party to the bilateral agreement, but MAY refuse connections from other sources, or ignore what might be complete agreements for this transaction from those sources, unless it has another obligation to accept such connections or complete agreements from them.

If a relay receives a complete agreement and acts on it in any way that might create obligations for other instances or relays (such as by forwarding it, whether or not it was obliged to do so), then it MUST fulfil all of its own obligations arising from any bilateral agreement or other arrangement, even if it received the complete agreement before the completion deadline, or from

an unexpected source. In the case of an early complete agreement, a relay acting on it has the same obligations as it would have had if it had received the complete agreement at the completion deadline.

Because the hashes of invitations are used, rather than the invitations themselves, each instance needs to have records of the contents of the original invitations, and needs to be able to recognize them by their hashes. It is RECOMMENDED that the hashes used are those of the most recent invitations communicated between the instances to update each other about their current medium-term public keys and lists of relays. Each hash MUST be of an invitation that listed at least three relays.

Example 5. After Example 1, Carlos owes Bob 100 USD. Carlos is assisting in an attempt to execute another payment via circulex. Carlos's paywise neighbour wants him to pay 500 USD, so if Bob is Carlos's talkwise neighbour again, then Bob agreeing to forgive Carlos's remaining debt would be insufficient to compensate for Carlos's obligation to his paywise neighbour. However, it can be used as part of the payment, with the remainder covered by Bob agreeing that he owes Carlos 600 NZD.

The following is an example of a bilateral agreement that encodes such an arrangement. It would be part of a message that Carlos's instance sends to Bob's instance, and that message would establish that Carlos's instance is the paywise peer. Such a bilateral agreement would begin with the time labels and invitation hashes, such as:

```
0x40000000573522aa3a7b2cd4
0x7be3579ba7231c35f2d257ea4e64973f
  8a593afd3860b6c9a205d53ce736ee6d
  539a1c5f69904540f56217685392a544
  081ce12d2b35031a70690592ab95e31c
0x40000000573522c22a8e8aec
0x64d340cdf78ebf92489fbdbcb698a2a7
  96ffc58d5b35e3757c7e686d67cad606
  3652ae05f7756697eda1305479d567ba
  ab37480854c043224be707fd08dc8746
```

(represented here in hexadecimal), and be followed by the UTF-8 representation of a JSON array encoding the payments discussed above, such as:

```
[
  {
    "paymentType": "record",
    "account": {
      "myUser": "root",
      "yourUser": "root",
```

```
      "currency": {
        "unit": "USD",
        "denominator": 100
      }
    },
    "numerator": 10000
  },
  {
    "paymentType": "record",
    "account": {
      "myUser": "root",
      "yourUser": "root",
      "currency": {
        "unit": "NZD",
        "denominator": 100
      }
    },
    "numerator": 60000
  }
]
```

5 Messages

5.1 Location request

An instance (say, Alice's) might sometimes find itself unable to communicate with a peer (say, Bob's instance) because it doesn't have a current usable instance location for Bob's instance. If Alice's instance is still able to communicate with a peer of Bob's instance (say, Carlos's instance), then Alice's instance can send a *location request* to Carlos's instance. This is a request for Carlos's instance to send a location report to Bob's instance, telling Bob's instance how to reestablish communication with Alice's instance.

A location request consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`lcrq`" (`0x6c637271`).

remaining bytes
: An instance identity, specifying the instance to which the requested

location report is to be sent.

The recipient of a location request SHOULD generally, where possible, respond by sending to the requested destination a location report whose instance identity identifies the sender of the location request and contains, in its `locations` member, all currently known usable instance locations for that sender, including one corresponding to the location from which the request was sent. However, instances MUST implement policies that protect themselves and their peers from any denial-of-service attacks that make use of location requests.

If an instance doesn't know its own public IP address or port, but can communicate with a peer, then it can attempt to learn an instance location for itself by sending its peer a location request, identifying itself in the instance identity.

5.2 Location report

A *location report* is a message sent in response to a location request. It's sent by the recipient of the request to the instance specified in the request. It consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`lcrp`" (`0x6c637270`).

remaining bytes
: An instance identity identifying the sender of the request corresponding to this report, listing at least one currently usable instance location in the value of the `locations` member.

5.3 Invitation

An *invitation* is a message that can be used by an instance to inform a peer of its currently preferred medium-term public key and set of relays, and its desired limit on the bandwidth used by incoming messages. It consists of the concatenation of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`nvtn`" (`0x6e76746e`).

bytes 12 **to** 43
: A public key, specifying the medium-term public key that the sender wants the recipient to use (for the time being) for generating temporary public keys for the sender, as described in Section 3.5.

bytes 44 **to** 47
: A bandwidth limit. The class of messages this limit relates to consists of the following messages sent by the recipient of the invitation to its sender:

  - location requests,
  - invitations,
  - statements,
  - relay offers,
  - relay rejections,
  - hints,
  - partial agreements, and
  - missing information requests for any of the above;

  together with the following messages sent to a typical one of the sender's relays (even if the sender is not acting as one of its own relays):

  - location requests sent by the recipient of this invitation in which the instance identity identifies the sender of the invitation, and
  - pings and complete agreements sent by the recipient's relays.

remaining bytes
: The UTF-8 representation of a JSON array of instance identities, specifying the list of relays the sender wants to use from now on. This list SHOULD include the sender's instance itself, so that it can directly send complete agreements to (and receive them from) its peer's relays.

When an instance wants to invite another instance to be its peer, it MUST send an invitation, to inform that peer of its current medium-term public key and set of relays, and its desired bandwidth limit. If the recipient also wants to establish that relationship, then it MUST respond with its own invitation.

As long as an instance wants to maintain the relationships it has with its established peers, it MUST send new invitations to them whenever its current set of preferred relays changes, and whenever it wants to change its medium-term public keys, and SHOULD send new invitations whenever its

desired bandwidth limit changes significantly. An instance MAY use different sets of relays and different bandwidth limits with different peers, and MUST NOT use the same medium-term public key with different peers, or reuse an old one with the same peer.

Instances SHOULD respect the bandwidth limits requested by their peers, planning ahead when sending messages that might require future use of the bandwidth to that peer and its relays.

During times of high load, instances MAY defer sending invitations; this will temporarily inhibit their ability to form new mutually satisfactory bilateral agreements with their peers.

5.4 Statement

A *statement* is a message used for communicating the sender's beliefs and intentions regarding changes to the balance of obligations (in a particular currency) between a member of the sender's instance and a member of the recipient's instance. It consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`stmt`" (`0x73746d74`).

bytes 12 **to** 23
: A time label specifying the beginning of the period of time this statement relates to.

bytes 24 **to** 35
: A time label specifying the end of the period of time this statement relates to.

bytes 36 **to** 39
: A number, τ, indicating the number of bilateral agreements that permanently affected the relevant balance and had a reversal deadline strictly before the end of the specified period, but not strictly before the beginning of the period.

bytes $32j + 8$ **to** $32j + 39$ **(for each** j **with** $1 \leq j \leq \tau$**)**
: The public key associated with such a bilateral agreement, generated as described in Section 5.12. Each such public key MUST appear in this list.

remaining bytes

A balance summary, specifying the account this statement relates to. The period specified in bytes 12 to 35 of this statement is the relevant period for the purposes of the `openingBalance` and `closingBalance` members of the balance summary.

If an instance wants to establish a relationship (in a particular currency) between one of its members and a member of another instance (for example, on behalf of its member, extending credit in that currency to the other instance's member), then it SHOULD send a statement to the other instance, communicating its intended upper and lower limits (if any) on the balance. If the recipient is also willing to establish that relationship, then even if it does not extend credit on behalf of its member, it SHOULD respond with its own statement in which the balance summary's `account` member specifies the same currency as in the statement it received, and has the values of the `myUser` and `yourUser` members reversed.

The recipient of a statement that specifies a nonzero credit limit will then be in a better position to determine which paywise partial agreements are worth forwarding to the sender of the statement, and which will be rejected because they would violate, or risk violating, the stated credit limit. Similarly, if the sender of a statement uses it to communicate a desired upper limit on debts owed by its member to the recipient's member, (or a desired lower limit on debts owed in the other direction), this can aid the recipient in determining which talkwise partial agreements to send. Any information conveyed in the `transactionSizes` or `maximumOutstanding` members of the balance summary might also be useful in determining which partial agreements are more likely to be accepted.

Another important factor in determining which partial agreements to send is the current balance of obligations between the members. Typically, the instances will independently come to the same conclusion regarding which of their bilateral agreements result in payments being executed, and which don't, based on which complete agreements their relays report having received, and when they report having first received them. Indeed, the protocol ensures that there are strong incentives against any course of action that might lead to disagreement between well-behaved peers with well-chosen deadlines and sets of relays.

However, it's theoretically possible that the talkwise instance might believe that it's obliged to execute a payment, while the paywise instance believes that no such payment need be made. Once such a disagreement has been detected, it will probably be easy to resolve amicably. The opposite disagreement is more serious, but it can only occur if at least half of the

relays belonging to one of the instances are offline or misbehaving at some point during the period of time in which the transaction's outcome is in doubt, or if the bilateral agreement's deadlines were poorly chosen, or the peers themselves misbehave.

In order to detect either kind of disagreement, instances SHOULD regularly send statements ensuring that their peers have complete information about their beliefs about changes to relevant balances; they SHOULD also check that the statements they receive match their own beliefs about those changes. In addition, each such update SHOULD include the sender's currently desired transaction and balance limits.

Note that a statement requires an exact list of bilateral agreements that permanently affected the relevant balance, and have reversal deadlines during the relevant period. For this reason, instances MUST NOT send statements in which the end of the specified period is in the future.

Finally, if an instance wants to end the relationship in a particular currency between one of its members and a member of another instance, then it needs to do two things. First, it MUST ensure that the balance becomes zero, and that there are no outstanding bilateral agreements that might make the balance nonzero again. Then, it MUST send a statement covering all past transactions that permanently affected the balance and that have not been included in any statements it has previously sent; the value of the `balanceLimits` member of the balance summary included in that statement MUST include the `minimum` and `maximum` members, whose values MUST be `0`.

5.5 Relay offer

A *relay offer* is a message an instance can send its peer in order to offer to act as a relay for that peer, within certain limits. It consists of the following 28 bytes:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`rlfr`" (`0x726c6672`).

bytes 12 **to** 23
: A time label, indicating that the sender of this relay offer will generally assent to relay requests from its recipient if their expiry times are no later than this time.

bytes 24 **to** 27
: A bandwidth limit. The class of messages this limit relates to consists of the following messages sent by the recipient of this offer to its sender:

  - pings,
  - relay requests,
  - complete agreements, and
  - missing information requests for receipts and tallies;

  together with pings and complete agreements sent to the sender by any of the recipient's peers' relays, and location requests sent to the sender by any of the recipient's peers in which the instance identity identifies the recipient.

5.6 Ping

A *ping* is a message that can be used by an instance (either a primary instance or a relay) to assess the one-way communication latency from itself to a specific destination. It consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`ping`" (`0x70696e67`).

bytes 12 **to** 23
: A time label encoding the time at which the ping was sent.

remaining bytes
: Arbitrary data, used for padding the ping to the desired length.

Pings can be relatively short messages, but instances SHOULD pad them to the same length as typical complete agreements, so that the latency of a ping better reflects the latency of a complete agreement. Alternatively, the same effect MAY be achieved by padding or bundling messages in the encryption or transport layer.

Instances and relays MUST, in general, reply to each received ping with a corresponding pong, as quickly as possible. However, they SHOULD implement sufficient rate-limiting to protect themselves from denial-of-service attacks. Also, when under very high load, replying to pings is a lower priority

than forwarding relevant complete agreements, so instances MAY temporarily ignore incoming pings, but this MUST be reserved for extreme situations only, and instances MUST endeavour to avoid getting into such extreme situations.

5.7 Pong

A *pong* is a message sent in response to a ping. It consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`pong`" (`0x706f6e67`).

bytes 12 **to** 23
: The time label included in the ping to which this pong is a reply.

bytes 24 **to** 35
: A time label encoding the time at which the ping was received.

If the second time label in a pong encodes a time earlier than time encoded by the first time label, this indicates a discrepancy between the clock belonging to the sender of the pong and that belonging to the recipient. In such a case, the parties SHOULD attempt to correct the clock that is in error (or both clocks, if both are in error).

If, before this correction occurs, the recipient of the anomalous pong sends any latency report that includes a latency profile whose `destination` member specifies the sender of the pong, then it SHOULD include in the `profile` member a latency probability object whose `latency` member is negative; this will inform the recipient of the latency report that the clock discrepancy exists.

5.8 Relay request

A *relay request* is a message that can be used by an instance to request that its desired relays act as relays for a particular transaction. It consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`rlrq`" (`0x726c7271`).

bytes 12 **to** 23
: A time label, specifying the time beyond which it is no longer necessary to forward complete agreements as requested by this message.

bytes 24 **to** 71
: A transaction identifier, specifying the transaction whose complete agreements the sender wishes the recipient to forward. The completion deadline for the transaction MUST be earlier than the expiry time specified in bytes 12 to 23.

remaining bytes
: The UTF-8 representation of a JSON array of instance identities, specifying the destinations to which the sender of this message wishes the recipient to forward the complete agreements it receives (before the specified expiry time) for the specified transaction.

If the recipient of a relay request is willing to act as a relay as requested, then it MUST respond with a latency report corresponding to the relay request.

If an instance sends a relay multiple relay requests for the same transaction, then the requests are cumulative; if it has received a latency report corresponding to an earlier relay request, it need not include the same destinations in a later relay request for the same transaction, unless it wishes to extend the expiry time for those destinations.

Once a relay has agreed to a relay request by sending the corresponding latency report, it MUST expeditiously forward to the destinations specified in the latency report the complete agreements (for the specified transaction) that it receives in the period of time between the completion deadline of the transaction and the expiry time specified in the relay request (inclusive of the completion deadline, but exclusive of the expiry time), at least until each destination has two distinct complete agreements for that transaction. During this period of time, the relay MUST accept connections and messages from the specified destinations (and from its primary instance, the sender of the relay request), in order to forward any relevant complete agreements that they send.

Note, however, the obligations in Section 4.17 regarding complete agreements that the relay might receive before the completion deadline and choose to use anyway. If the relay receives and forwards a complete agreement before the transaction's completion deadline, then it MUST NOT assume that the recipients have not ignored it; instead, it MUST forward the complete agreement again at the completion deadline, and fulfil all of its other obligations, as if it had received the complete agreement at the completion deadline.

A relay MUST NOT agree to a relay request if the completion deadline has already passed, and it has (or might have) already failed to fulfil any of the obligations (to accept connections and messages, or to forward complete agreements) that it would have had if it had agreed to the relay request before the completion deadline.

If a relay receives the same complete agreement multiple times, it need not forward it again to destinations that have already received it; also a relay need not send a complete agreement back to a destination from which it received that complete agreement.

A relay that has agreed to a relay request MUST also send receipts to its primary instance for the first two distinct complete agreements (for the specified transaction) that it receives between the completion deadline and the expiry time.

A relay that receives more than two distinct complete agreements for the transaction during that period MAY forward a third one to any of the specified destinations, and MAY send its primary instance a receipt for a third one, but MUST NOT forward or send receipts for more than three.

Finally, after the latest expiry time specified in any latency report that a relay sent to its primary instance for a particular transaction, the relay MUST send the primary instance a tally, indicating the number of distinct complete agreements (for that transaction) for which it sent its primary instance at least one recipt.

5.8.1 Null relay request

Suppose an instance has received a relay offer from a peer. Before including that peer in its relay lists in invitations, it might want to determine the latency and reliability of communication from that potential relay to the instance's peers' relays. For this purpose, the instance can send a *null relay request*, which has the same form as a relay request, but slightly different rules.

In a null relay request, the transaction identifier has the completion deadline set to the expiry time of the request (rather than an earlier time), and every byte of the initiator's public key set to zero (`0x00`).

The recipient of a null relay request SHOULD respond to it with a latency report if it would generally assent to relay requests from the same sender that have:

- an equal or earlier expiry time,
- an equal or smaller number encoded in the first four bytes of the transaction identifier, and

- a list of destinations that's equal to or a subset of the list in the null relay request;

otherwise it MUST NOT send a corresponding latency report.

A relay that responds to a null relay request with a latency report isn't obliged to forward complete agreements or send receipts or tallies for that transaction.

5.9 Latency report

A *latency report* is a message sent in reply to a relay request, indicating assent to the request. It consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`ltnc`" (`0x6c746e63`).

bytes 12 **to** 23
: The time label included in the relay request to which this latency report is a reply.

bytes 24 **to** 71
: The transaction identifier included in the relay request.

remaining bytes
: The UTF-8 representation of a JSON array of latency profiles. The expiry time and transaction relevant to the embedded latency probability objects are specified by the time label and transaction identifier included in this latency report. The `destination` member of each latency profile is an element of the JSON array in the original relay request. The sender of this latency report SHOULD include a latency profile for each instance identity in that request, except for any destinations that it's currently unable to communicate with.

5.10 Relay rejection

If an instance's relays report that they're unable to reliably communicate with one or more relays that were listed in an invitation that a peer sent to the instance, then that instance will be unable to safely transact with that peer unless the peer sends another invitation — one which doesn't list the troublesome relays. To inform the peer about which relays are problematic,

the instance can send a *relay rejection*, which consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`rlrj`" (`0x726c726a`).

remaining bytes
: The UTF-8 representation of a JSON array of instance identities, listing the recipient's requested relays that the sender's relays are unable to reliably communicate with.

An instance SHOULD NOT send a relay rejection for a group of its peer's relays if the group of its own relays that report inadequately reliable communication with them is a minority of its own relays and is no larger than the problematic group of its peer's relays; instead, it SHOULD assess the reliability of those of its own relays that are reporting the problem, and consider replacing them.

5.11 Hint

A *hint* is a message that can simultaneously serve two purposes. First, it specifies a target for pathfinding for a transaction; and second, when an instance sends a hint to some of its peers, this indicates to them its willingness to have them as neighbours in that transaction.

A hint consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: Either the string "`thnt`" (`0x74686e74`), which specifies that this is a *talkwise hint* (that is, a hint that is to be sent to potential talkwise neighbours), or the string "`phnt`" (`0x70686e74`), which indicates a *paywise hint*, to be sent to potential paywise neighbours.

bytes 12 **to** 59
: A transaction identifier, specifying the transaction to which this hint relates.

bytes 60 **to** 123
: A signature of bytes 8 to 59 of this message, concatenated with the bytes from byte 124 to the end of the message, valid under the initiator's public key, as specified in the transaction identifier.

remaining bytes
: The UTF-8 representation of a target.

The initiator of a transaction MUST NOT create a signature of the form specified for a hint unless it wants instances involved in pathfinding for the transaction to attempt to create a path involving the details specified in the target. The initiator SHOULD create two hints for the transaction, one for the talkwise direction and one for the paywise direction.

An instance that creates or receives a paywise hint for a transaction it is willing to participate in SHOULD send it to peers it is willing to have as its paywise neighbour for that transaction, but MUST NOT do so unless:

- it is the initiator of the transaction, or
- both of the following are true:
  - it is not the instance specified by the `host` member of the target, and
  - it has received the hint from a peer it is willing to have as its talkwise neighbour.

The situation is symmetrical in the case of a talkwise hint — an instance sends the hint to potential talkwise neighbours, but only if it created the hint or received it from a potential paywise neighbour (and is not itself the target host).

During times of high load, an instance SHOULD prioritize those of the hints it receives that it judges most likely to result in successful transactions involving itself.

Receipt of a hint proves that there is at least a chain of communication from the initiator to the recipient; ideally it indicates that the chain of communication coincides with a chain of willing participants in the specified direction (either talkwise or paywise). However, it is possible that the chain of communication includes an instance that is trusted by neither the initiator nor the recipient, and which has no intention of extending any partial agreements it might receive for that transaction. Therefore, instances MUST NOT rely on receipt of a hint as proof of the existence of a chain of willing participants; instead, an instance that wants to increase the likelihood that

it is involved in a particular transaction SHOULD, whenever possible, send hints and partial agreements for that transaction to multiple peers.

It might seem pointless to create or forward a hint that is merely an empty JSON object; after all, it fails to specify a common target for pathfinding, and the partial agreements will *prove* the existence of chains of willing participants, whereas the hint can only be indicative. However, the hint, which only needs to be forwarded, might easily travel faster than the partial agreements, which need to be extended by each interested recipient. An instance that receives both the talkwise and the paywise hints for a transaction can forward the talkwise hint to the neighbour or neighbours it received the paywise hint from, and vice versa; then any instance that receives a hint from one direction and a partial agreement from the other will be much better informed about which peers to send partial agreements to. In this way, even hints with empty targets can facilitate pathfinding.

5.12 Partial agreement

A *partial agreement* is a message that can be used to incrementally construct a complete agreement for a transaction. A partial agreement consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`prtl`" (`0x7072746c`).

bytes 12 **to** 15
: A number, m, indicating the length of the partial agreement.

bytes 16 **to** $96m + 67$
: A signed payment path of length m.

remaining bytes
: A bilateral agreement whose deadline is after the completion deadline of the transaction specified in the signed payment path.

The last public key in the signed payment path MUST be a temporary public key generated in the way specified in Section 3.5; the sender acts as instance 1, the recipient as instance 0; the bytes from byte 16 to byte $32m + 35$, concatenated with the bytes from byte $96m + 68$ to the end of the partial agreement, serve as the data x.

A partial agreement can be created by the initiator of a transaction, or adapted by a recipient, who creates a new partial agreement by increasing the value of m and constructing a bilateral agreement they are willing to offer to one of their peers, adding the peer's temporary public key (generated as described above) to the signed payment path (along with the necessary additional signature). Instances MAY also add more of their own public keys to the signed payment path before the public key they generated for their peer (also adding the necessary signatures), in order, for example, to obscure their proximity to the initiator.

When deciding whether to create a signature of the form that can appear in a signed payment path, there are two major considerations:

- the provenance of the key under which the signature is valid, and
- the provenance of the following public key in the signed payment path.

Regarding the first, an instance MUST NOT create the signature unless one of the following is true:

- it generated the key itself (for example, as the initiator, or in order to pad the signed payment path with extra keys); or
- both of the following are true:
  - the key was generated as described above from the relevant parts of a partial agreement, and
  - the instance agrees to the bilateral agreement contained in that partial agreement.

Regarding the following public key, an instance MUST NOT create the signature unless one of the following is true:

- it generated the key itself, and it is the only entity that knows or can calculate a discrete logarithm (base B) of that public key; or
- all of the following are true:
  - the key was generated as described above from the relevant parts of a partial agreement that the instance is preparing to send,
  - the instance agrees to the bilateral agreement contained in that partial agreement, and
  - the instance reasonably believes that the recipient of the partial agreement is the only entity that knows or can calculate a discrete logarithm (base B) of that public key.

Additionally, an instance SHOULD NOT create such a signature if it would be unusable as part of a complete agreement, either because a relevant deadline has already passed, or because the signed payment path would be too long (taking into account that the last public key in the signed payment path of a complete agreement has to be the same as the initiator's public key).

5.13 Complete agreement

A *complete agreement* is a message used to indicate which path (if any) was chosen for a particular transaction. A second complete agreement for the same transaction can be used to reverse the execution of the transaction. A complete agreement consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`cplt`" (`0x63706c74`).

bytes 12 **to** $96m + 63$
: A signed payment path whose length, m, is equal to the number encoded in the first four bytes of the transaction identifier. The last public key in the path MUST be the same as the initiator's public key.

bytes $96m + 64$ **to** $96m + 127$
: A valid signature, under the initiator's public key, of the bytes from byte 8 to byte $32m + 63$ of this message.

Two complete agreements are considered to be the same as each other if they agree on the bytes from byte 8 to byte $32m + 63$, even if they disagree on the circulex version information or one or more of the signatures.

The initiator of a transaction MUST NOT create a signature suitable for use as the final signature in a complete agreement except when doing so for one of the purposes described below. Note that it is not compulsory for implementations to be able to implement (or even distinguish between) all of the following uses of complete agreements. Regarding recognition of complete agreements, an instance that merely wants to participate in a transaction only needs to discern which complete agreements are valid ones (as defined above) for that transaction, and need not concern itself with the circumstances under which the complete agreement was created. Regarding the creation of complete agreements, an instance that wants to be the initiator of a transaction only needs to be able to produce complete agreements in accordance with the rules of Section 5.13.1 (though also being able to produce

them in accordance with the rules of Section 5.13.3 might encourage its peers to be more coöperative with it in future); being able to produce complete agreements suitable for the other circumstances is OPTIONAL.

5.13.1 Executing a transaction

A typical complete agreement includes a signed payment path with keys and signatures belonging to multiple instances; it is constructed by the initiator in order to execute the transaction along that payment path.

In such a case, the initiator MUST NOT create a signature that can be used as the final signature in the complete agreement unless:

- it wants to execute the transaction along that payment path, and is able to fulfil its payment obligations for that payment path;
- it can and will ensure that the complete agreement reaches (and cannot legitimately be ignored by) its talkwise partner's relays before the deadline specified in the relevant bilateral agreement with that partner; and
- it has not already created such a signature suitable for a different complete agreement for the same transaction.

5.13.2 Reversing a transaction

If a complete agreement has caused the execution of a transaction, but the relevant reversal deadline hasn't yet passed, the initiator can reverse the transaction by using a *trivial complete agreement* — that is, a complete agreement for which the initiator has sole control over all of the secrets associated with all of the public keys in the signed payment path. (Note that only the initiator will be able to know with certainty that the complete agreement is trivial, unless it deliberately proves this fact to others.)

In such a case, the initiator MUST NOT create a signature that can be used as the final signature in the complete agreement unless:

- it wants to reverse the original transaction; and
- it can and will ensure that the trivial complete agreement reaches (and cannot legitimately be ignored by) the relays of its paywise partner (in the original transaction) before the reversal deadline specified in the relevant bilateral agreement with that partner.

It is also possible to use a nontrivial complete agreement for this purpose, as long as it differs from the original complete agreement in the sense defined above. However, this is NOT RECOMMENDED; an instance that discovers that two different nontrivial complete agreements exist for the same transaction (such as any instance that controls a key in each signed payment path) might suspect that one of the complete agreements was the result of the initiator's instance malfunctioning or leaking a secret.

Furthermore, in order to prevent the execution of the transaction specified by the second nontrivial complete agreement, the initiator MUST ensure that the original complete agreement reaches (and cannot legitimately be ignored by) the relays of its paywise partner (in the second complete agreement) before the reversal deadline specified in the relevant bilateral agreement with that partner.

5.13.3 Committing to non-execution

If the initiator of a transaction simply refrains from creating any complete agreement for that transaction, the other potential participants will need to wait until the expiry of the deadlines in their bilateral agreements before they can rely on the non-execution of the transaction; this might unnecessarily tie up their funds, making them less inclined in future to make bilateral agreements for transactions that appear to originate from the same initiator.

To avoid this, the initiator might want to promptly commit to the non-execution of the transaction, as soon as it has decided it will not execute it. This can help the other potential participants in the transaction by assuring them that they don't need to remain prepared for the possibility that they will have to participate in the execution of that transaction.

In order to make this commitment, the initiator can create a trivial complete agreement and send it to its peers (via their relays). The initiator MUST NOT create a signature suitable for use as the final signature in such a complete agreement unless it wants to irrevocably commit to the non-execution of the transaction.

The initiator MAY send such a complete agreement to its peers before the completion deadline, in order to inform them as soon as possible of their non-participation in the transaction. However, some relays that receive the complete agreement before the completion deadline might ignore it, so the initiator SHOULD resend it at or after the completion deadline, in order to ensure that all interested instances are aware of their non-participation in the transaction.

5.13.4 Partially executing a transaction

It is possible for the initiator of a transaction to own another, intermediate, public key (or contiguous sequence of public keys) in a signed payment path, both preceded and followed by at least one public key controlled by another instance. The initiator might wish to arrange for this to happen if, for example, pathfinding for the transaction has discovered an arbitrage opportunity.

It is then possible for the initiator to arrange for the partial execution of the transaction, executing some of the bilateral agreements in the signed payment path after the deadlines in the others have expired. The initiator might want to do this if, for example, it has not created a complete agreement before the deadline in the talkwise bilateral agreement associated with its primary public key in the signed payment path (that is, the public key specified in the transaction identifier).

In such a case, the initiator MUST NOT create a signature suitable for use as the final signature in the complete agreement unless:

- the deadline in the paywise bilateral agreement associated with its intermediate public key has already passed;
- it wants to execute the remainder of the transaction (talkwise from its intermediate public key, and paywise from its primary public key), and is able to fulfil its payment obligations for that part of the transaction;
- it can and will ensure that the complete agreement will reach (and cannot legitimately be ignored by) the relays of its partner in the talkwise bilateral agreement associated with its intermediate public key, before the deadline in that bilateral agreement; and
- it has not already created a signature suitable for a different complete agreement for the same transaction.

Note that, if the search direction in the signed payment path is `pyws`, it is much simpler (and probably less prone to error) if the initiator removes the public keys in the signed payment path after its intermediate public key (retaining only the relevant signatures) and uses the result to construct a complete agreement for use as in Section 5.13.1.

5.13.5 Partially reversing a transaction

Similarly, suppose the initiator of a transaction has executed the transaction using a signed payment path in which it controls an intermediate public key.

And suppose the initiator wants to partially reverse that transaction — perhaps it wants to reverse the purchase for which the transaction was originally intended, without reversing the execution of an arbitrage opportunity that was found during pathfinding.

The initiator can achieve this by creating — at a specific time — a trivial complete agreement (or any complete agreement different from the one originally used to execute the transaction, though Section 5.13.2's caveats about nontrivial complete agreements apply here, too), and sending it to the paywise neighbour of its intermediate public key.

In this case, the initiator MUST NOT create a signature suitable for use as the final signature of the second complete agreement unless:

- the reversal deadline in the talkwise bilateral agreement associated with its intermediate public key has already passed;
- it wants to reverse the portion of the transaction that is paywise of its intermediate public key and talkwise of its primary public key; and
- it can and will ensure that the second complete agreement will reach (and cannot legitimately be ignored by) the relays of its partner in the paywise bilateral agreement associated with its intermediate public key, before the reversal deadline specified in that bilateral agreement.

Note that in both this case and that of partial execution, the part of the transaction that is executed and remains executed is the part that is paywise of the initiator's primary public key and talkwise of its intermediate public key.

A related note is that if the initiator wants to reverse a transaction that was partially executed, the rules to follow are usually those of Section 5.13.2, not those of this section. The exception is if the initiator controls multiple intermediate public keys, with other instances' public keys before, after, and between them; in that case, the initiator might at first want to execute only part of the signed payment path constructed during pathfinding, and then later want to reverse only part of that partially executed transaction; then this section is the applicable one.

5.14 Receipt

A *receipt* is a message that a relay can send to its primary instance to inform that instance about when the relay received a particular complete agreement. It consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`rcpt`" (`0x72637074`).

bytes 12 **to** 23
: A time label, specifying the earliest time at which the relay sending this receipt received the excerpted complete agreement. If the relay first received the complete agreement before the completion deadline, and chose not to ignore it, then this time label can encode any time between the time at which the relay first received the complete agreement and the completion deadline (inclusive).

remaining bytes
: The body of a complete agreement, from byte 12 of the complete agreement to its end.

When an instance receives a receipt, it can reconstruct the complete agreement on which it was based by concatenating the circulex version information, the string "`cplt`", and the data from byte 24 to the end of the receipt; therefore, an instance is considered to have received a complete agreement as soon as it has received a corresponding receipt.

5.15 Tally

A *tally* is a message sent by a relay to a primary instance to confirm to that instance the number of distinct complete agreements for a particular transaction for which the relay sent the instance receipts. It consists of the following 64 bytes:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`taly`" (`0x74616c79`).

bytes 12 **to** 59
: A transaction identifier.

bytes 60 **to** 63
: The number of distinct complete agreements for the specified transaction for which the relay sent the instance receipts.

A relay MUST NOT send to any instance a tally for a transaction before the expiry time specified in any latency report it sent to the instance regarding that transaction. After a relay has sent a tally to an instance, it MUST NOT send that instance any receipts for any complete agreements for that transaction, except those complete agreements for which it has already sent the instance receipts.

5.16 Missing information request

A *missing information request* is a message that can be used to request information that the sender hasn't received. It begins with the following data:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: The string "`msng`" (`0x6d736e67`).

The body of a missing information request depends on the type of information being requested.

The recipient of a missing information request MUST, in general (assuming that it's both possible and practical to do so), respond with the requested information, but only if the requester would ordinarily have access to that information in the normal course of operation of the circulex protocol. Each instance SHOULD ensure that its policy regarding responding to missing information requests doesn't make it unnecessarily vulnerable to denial-of-service attacks.

Depending on the type of information requested, and, in some cases, on the preferences of the recipient of the request, the response can be sent either as a plain message of the type requested, or as a quoted message of that type (or multiple messages of either type or both, if the information can be spread across multiple messages). In some cases, such as that of a missing tally from a relay, the relay can simply resend the tally without any undesirable effects. In other cases, such as that of a missing invitation, simply resending the invitation would convey to its recipient that its contents reflect its sender's current preferences; if it accurately reflects the sender's preferences, then the sender SHOULD simply resend the invitation; otherwise, the sender will prefer to respond to the request with a quoted invitation.

Implementations MUST take care in determining when to respond to missing information requests with quoted messages, and when to respond with unquoted messages. The sender of a missing information request MUST be

able to comprehend both quoted and unquoted responses, regardless of the type of message requested.

5.16.1 Request for invitation

If an instance is trying to interpret a bilateral agreement and finds it doesn't have a record of the invitation corresponding to one of the hashes, it can request that the other party to the bilateral agreement sends the invitation. In such a case, the body of the missing information request is as follows:

bytes 12 **to** 15
: The string "`nvtn`".

bytes 16 **to** 79
: The hash of the invitation being requested.

If an instance has lost its copy of one of its own invitations that it sent in the past, it can use such a request to ask the recipient to send back a copy of the invitation, but if the peer is able to fulfil that request, it MUST do so using a quoted invitation, not an unquoted one.

5.16.2 Request for statements

If an instance lacks information from one of its peers regarding a particular period of time during which a particular account was (or might have been) active, it can request the relevant statements from that peer. In such a case, the body of the missing information request is as follows:

bytes 12 **to** 15
: The string "`stmt`".

bytes 16 **to** 27
: A time label specifying the start of the period of time for which statements are requested.

bytes 28 **to** 39
: A time label specifying the end of the period of time for which statements are requested. This MUST specify a time after that specified by the previous time label, but before the time at which the request is sent.

remaining bytes
: An account specifier, specifying the account for which statements are requested.

A valid response to such a request consists of one or more quoted or unquoted statements that together cover the specified period of time, and whose account specifiers specify the same currency, but have the values of the `myUser` and `yourUser` members reversed.

5.16.3 Request for hint

Suppose an instance has received a partial agreement from one of its peers, and it is interested in building on that partial agreement in an attempt to participate in that transaction. But suppose also that the instance hasn't received a relevant hint for the transaction, and wants the information that such a hint might contain, in order to make a better-informed decision about whether and how to build on the partial agreement. In such a case, the instance can request a hint from the peer that sent the partial agreement; it does so by sending a missing information request with the following body:

bytes 12 **to** 15
: Either the string "`thnt`" or the string "`phnt`", respectively requesting either a talkwise hint (from a peer that sent a talkwise partial agreement) or a paywise hint (from a peer that sent a paywise partial agreement).

bytes 16 **to** 63
: The transaction identifier for the transaction whose hint is being requested.

An instance that receives such a request, but lacks the means to respond to it, can forward the request back to the relevant peers of its own, from which it received the bases of any partial agreements it had sent to the requester; if it receives a response, it can then forward that to the original requester.

Requests for hints MAY also be sent in other circumstances, but the benefit of doing so might be negligible.

5.16.4 Request for partial agreement

Suppose an instance receives a statement from one of its peers, but has no record of the bilateral agreement associated with one of the public keys listed in the statement. The instance can use a missing information request to ask its peer for the original partial agreement that contained the relevant bilateral agreement. In such a case, the body of the missing information request is as follows:

bytes 12 **to** 15
: The string "`prtl`".

bytes 16 **to** 47
: The public key for which the associated partial agreement is requested.

The recipient of such a request SHOULD respond with a quoted partial agreement in which the last public key matches the public key in the request, but only if that partial agreement was originally sent (in either direction) between the sender and the recipient of the request.

5.16.5 Request for receipts

Suppose an instance receives a tally from one of its relays, and the tally indicates that the relay sent more receipts for that transaction than the instance received. Then the instance can use a missing information request to ask its relay to resend the receipts for that transaction. The body of such a missing information request is as follows:

bytes 12 **to** 15
: The string "`rcpt`".

bytes 16 **to** 63
: The transaction identifier of the transaction whose receipts are being requested.

The recipient of such a request SHOULD respond by resending the receipts (either quoted or unquoted) that it sent to the requester for that transaction.

5.16.6 Request for tally

Suppose an instance has received from one of its relays at least one latency report for a particular transaction, and suppose that the instance has allowed a reasonable length of time for communication latency from that relay after the last expiry time in any relay request it sent to the relay, but it hasn't yet received a tally for that transaction from the relay. The instance can use a missing information request to ask the relay to resend the tally. In such a case, the body of the missing information request is as follows:

bytes 12 **to** 15
: The string "`taly`".

bytes 16 **to** 63
: The transaction identifier of the transaction for which the tally is requested.

The recipient of such a request SHOULD respond by resending (or sending) a tally (quoted or unquoted) for the specified transaction, but only if it sent at least one latency report for the transaction to the requester, and the latest expiry time in any such latency report has already passed.

5.17 Quoted message

A *quoted message* can be used by an instance to give one of its peers information that the peer has either lost or never received in the first place. Quoted messages are useful when the sender doesn't want to convey the same intention as would be communicated if it simply resent the message, unquoted.

A quoted message consists of the following data in the following order:

bytes 0 **to** 7
: The circulex version information.

bytes 8 **to** 11
: Either the string "`qtme`" (`0x71746d65`) or the string "`qtyu`" (`0x71747975`), indicating, respectively, whether the enclosed message was originally sent by the sender of this message to its recipient, or by the recipient of this message to its sender. The latter case has implications for the interpretation of the `myUser` and `yourUser` members of any account specifiers included in the message.

remaining bytes
: The entire message being quoted, including its circulex version information.

References

[1] Daniel J. Bernstein. *TAI64, TAI64N, and TAI64NA*. URL: `https://cr.yp.to/libtai/tai64.html` (visited on 2016-05-12).

[2] Daniel J. Bernstein et al. *EdDSA for more curves*. July 4, 2015. URL: `http://ed25519.cr.yp.to/eddsa-20150704.pdf` (visited on 2015-10-22).

[3] Scott Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. RFC 2119.

[4] Tim Bray, editor. *The JavaScript Object Notation (JSON) Data Interchange Format*. March 2014. RFC 7159.

[5] David Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. May 2008. RFC 5280.

[6] Michelle Cotton et al. *Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry*. August 2011. RFC 6335.

[7] Morris J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Federal Information Processing Standards Publication 202. Information Technology Laboratory, National Institute of Standards and Technology, August 4, 2015. DOI: 10.6028/NIST.FIPS.202. URL: https://www.nist.gov/customcf/get_pdf.cfm?pub_id=919061 (visited on 2015-10-15).

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Journal of the Association for Computing Machinery* 32.2 (April 1985), pages 374–382. URL: http://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf (visited on 2015-07-10).

[9] Olafur Gudmundsson. *Adding Acronyms to Simplify Conversations about DNS-Based Authentication of Named Entities (DANE)*. April 2014. RFC 7218.

[10] Henry. *A Brief Tour of FLP Impossibility*. August 13, 2008. URL: http://the-paper-trail.org/blog/a-brief-tour-of-flp-impossibility/ (visited on 2015-07-10).

[11] Robert M. Hinden and Stephen E. Deering. *IP Version 6 Addressing Architecture*. February 2006. RFC 4291.

[12] Simon Josefsson. *The Base16, Base32, and Base64 Data Encodings*. October 2006. RFC 4648.

[13] Joyce Kim. *Safety, liveness and fault tolerance—the consensus choices*. December 5, 2014. URL: https://www.stellar.org/blog/safety_liveness_and_fault_tolerance_consensus_choice/ (visited on 2015-07-10).

[14] Joyce Kim. *Stellar Consensus Protocol: Proof and Code*. April 8, 2015. URL: https://www.stellar.org/blog/stellar-consensus-protocol-proof-code/ (visited on 2015-07-10).

[15] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (July 1982), pages 382–401. URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf (visited on 2015-07-09).

[16] Gary Scott Malkin, editor. *Internet Users' Glossary*. August 1996. RFC 1983.

[17] David Mazières. *The Stellar Consensus Protocol. A Federated Model for Internet-level Consensus*. Version July 14, 2015 DRAFT. July 14, 2015. URL: https://www.stellar.org/papers/stellar-consensus-protocol.pdf (visited on 2015-07-16).

[18] Jan Michelfeit. "Security and Routing in the Ripple Payment Network". Master's thesis. Brno: Masaryk University, 2011. URL: https://is.muni.cz/th/139865/fi_m/dp_139865.pdf?lang=en (visited on 2017-04-20).

[19] Dr. David L. Mills et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. Edited by Jim Martin. June 2010. RFC 5905.

[20] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. January 2012. RFC 6347.

[21] Ripple Communications. *About Ripple*. URL: https://classic.ripplepay.com/about/ (visited on 2015-07-10).

[22] Stellar. *Explainers*. URL: https://www.stellar.org/learn/explainers/ (visited on 2015-07-10).

[23] Randall R. Stewart, editor. *Stream Control Transmission Protocol*. September 2007. RFC 4960.

[24] The Unicode Consortium. *The Unicode Standard*. URL: http://www.unicode.org/versions/latest/ (visited on 2015-10-16).

[25] François Yergeau. *UTF-8, a transformation format of ISO 10646*. November 2003. RFC 3629.