The Circulex Protocol
DRAFT

T. J. M. Makarios

Revision: `7a5af96`
January 26, 2023

Abstract

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

This is currently a draft.

Contents

1 Introduction

1.1 Payments through chains of trust

There's a long-standing system of money transfer called *hawala* [31], which uses networks of trust relationships to transfer value quickly over long distances, often without any physical goods being moved at all. In some ways, payments between customers of different banks (including foreign exchange transactions) are executed in a hawala-like way, but hawala is considered to be "informal", and it relies more on trust relationships, and less on the legal enforceability of contracts — the former being more reliable in many parts of the world.

A project now known as Rumplepay had the idea of creating an automatic hawala-like network over the internet [33].

It's worth looking at an example of the kind of transaction such a system might enable:

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. The cyclic nature of such transactions is discussed and illustrated in Section 2.2 (particularly Figure 2.6) of Jan Michelfeit's thought-provoking Master's thesis [20] about transaction routing in a Rumplepay-like network (then referred to as *Ripple*). 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.

It's worth noting that the trust relationships don't have to be symmetrical. In Example 1, Bob already trusts Carlos to repay the 500 USD debt; Carlos need not trust Bob to pay 400 USD, because if they have a dispute about the execution of the transaction, 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 between them about the execution of the transaction, then he can simply refuse to pay her the 400 USD.

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 [36] solves these problems by maintaining a single authoritative ledger, which records the trust relationships and outstanding debts between entities. It aims to distribute authority over the ledger among many independent entities, but as of November 2022, it's still the case that the safety of the entire Stellar network could be compromised by the actions of nodes in a single jurisdiction, perhaps under legal duress [37]. Further, even if it successfully decentralizes authority more than it is at present, Stellar 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* [14], 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 [9]. 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 [32]. Such protocols can be designed to prevent disagreements between well behaved participants even if a certain proportion of the participants are acting maliciously [19].

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 agreement*) 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. (Note that although circulex relays might often also act as libp2p circuit relays, the roles are distinct.) 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 par-

ticipant 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 trans-

action, 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 bial-

teral 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.

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, it asked him whether he wanted it to keep his IP address private, noting the tradeoffs involved. Because Bob lives in a country with a great degree of freedom of association [40], he allowed the app to reveal his IP address freely. The app also allowed Bob to set an approximate limit on the bandwidth he wanted to let it use.

Alice got her computer to display her *participant identity* as a two-dimensional barcode, which Bob scanned with his phone's camera. The app

asked Bob how he wanted it to refer to the owner of the particpant identity, and he typed "Alice". 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 the port used by circulex, 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 port. Alice's and Bob's applications used core libp2p protocols to enable Bob's app to learn its public IP address and port, and to confirm that incoming connections to that address and port were possible.

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 tested its latency to Alice's computer by sending *pings* and receiving *pongs* in response. It then 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 then sent new invitations, which included the new relays, to his peers.

Bob's app sent an *indicative relay request* to Alice's and Carlos's applications (in their roles 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) sent *indicative relay peering messages* to his peers' relays, and received similar messages from them (perhaps as a result of indicative relay requests from Alice and Carlos), signifying a willingness to allocate bandwidth to their roles as peers' relays. Bob's relays then started to measure the latency to his peers' relays by sending pings to them and receiving pongs in response. Alice's and Carlos's applications then responded to the indicative relay requests by each sending a *latency report* to Bob's app.

These relationships are maintained by sending fresh invitations, statements, relay offers, pings, pongs, indicative relay requests, indicative relay peering messages, and latency reports, as necessary. Occasionally, if two circulex instances that need to communicate are unable to do so, *relay rejections* might be required, which an instance can use to inform its peer that the instance's relays can't reliably 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 payment is expected to be made via NZD and 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 that the payment is expected to be received via KRW and to have a value of 450,000 KRW; this hint performs a pathfinding function similar to that of the talkwise hint, but in the other direction.

Transaction relay requests and related messages As soon as an instance that's willing to participate receives a hint for Alice's transaction, it can send *transaction relay requests* to its relays. A transaction 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 transaction 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 sends *transaction relay peering messages* to each of Alice's relays, which send similar messages back (perhaps in response to transaction relay requests from Alice's application). These messages indicate willingness to receive complete agreements for the transaction.

Carlos's instance then 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 transaction 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 sends and receives similar messages, in its role as Bob's other relay.

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 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.

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 BCP 14

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 BCP 14 [6] [5] [15] when, and only when, they appear in all capitals, as shown here.

3.2 ASN.1

The specification of circulex is given below using Abstract Syntax Notation One (ASN.1) [41], including the use of information objects [42] and constraints [43]. Except where otherwise specified, the Canonical Octet Encoding Rules (COER) [44] are used.

The ASN.1 module specifying circulex begins as follows:

```
Circulex
    DEFINITIONS
    AUTOMATIC TAGS ::=

BEGIN

    IMPORTS Nanosecond FROM TAI64N;

    -- Generally useful things
    LimitedString       ::= OCTET STRING (SIZE (0 .. 255))
    NonnegativeInteger  ::= INTEGER (0 .. MAX)
    PositiveInteger     ::= INTEGER (1 .. MAX)
    TimeLabel           ::= Nanosecond
```

The `TAI64N` module is defined in Section 3.3; the remaining ASN.1 fragments in this document are from the rest of the `Circulex` module.

Each `LimitedString` MUST contain the BOCU-1 encoding (without reset bytes) of a Unicode string [34].

Most enumerated and choice types in the ASN.1 specification of circulex use the identifier `experimental` to reserve room for experimenting with possible extensions to the protocol before standardizing them, if the experiment is successful (see Internet Best Current Practice 82 [27]). The recipient of a message that uses this facility MUST NOT rely on any interpretation of the experimental part of the message unless it has confirmed that the creator of that part of the message intended that interpretation. However, other parts of the message might still be useful to the recipient. For example, the recipient of a hint can still obtain useful information from other parts of the hint even if its target currency is experimental, and the recipient has no knowledge of the experiment; if the recipient wishes to take part in that transaction, it is also still useful to forward the hint to potential neighbours in the transaction.

Data associated with an experimental alternative is always in the following form:

```
    Experimental    ::= SEQUENCE {
        code    OCTET STRING (SIZE (6)),
        name    LimitedString,
        data    OCTET STRING
    }
```

The elements have the following meanings:

`code`
: A six-byte code generated for the experiment randomly or pseudorandomly (from a uniform distribution over all possible six-byte codes). If the experiment changes in a way that breaks compatibility with earlier versions of the experiment, a new six-byte code MUST be generated for the newly-altered experiment.

`name`
: A short name for the experiment, which SHOULD NOT be empty. Incompatible iterations of an evolving experiment can use the same name.

`data`
: Any data required by the experiment in the relevant context. For example, the data might be the parameters (if any) of an experimental signature scheme.

3.3 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 [21] for this purpose.

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.

Circulex uses TAI64N labels [3] as *time labels*. The ASN.1 specification of TAI64N labels is:

```
TAI64N
    DEFINITIONS
    AUTOMATIC TAGS ::=

BEGIN
```

```
    Second ::= INTEGER (0 .. 9223372036854775807)

    Nanosecond ::= SEQUENCE {
        second      Second,
        nanosecond  INTEGER (0 .. 999999999)
    }

END
```

Note that the COER encoding of a `Second` or `Nanosecond` is identical to the external TAI64 or TAI64N format encoding of the same TAI64 or TAI64N label, respectively.

3.4 Hashes

A `HashAlgorithm` specifies an algorithm to be used for hashing, as well as the length of its output to be used.

```
    -- Hashes
    HashAlgorithm ::= SEQUENCE {
        algorithm       OCTET STRING,
        outputLength    PositiveInteger
    }
```

Its elements have these meanings:

`algorithm`
: The code of the hash algorithm, as defined by the canonical multicodec table [24].

`outputLength`
: The number of bytes of its output to be used. If the algorithm lacks a canonical output length, or this number differs from the canonical length, then a prefix of the output is to be used, its length being the number of bytes specified here.

```
    Hash ::= SEQUENCE {
        algorithm   OCTET STRING,
        body        OCTET STRING
    }
```

The elements of a `Hash` are:

`algorithm`
: The canonical multicodec code of the algorithm used to generate this hash.

`body`
: The value of the hash itself.

Note that the multiformats unsigned-varint encoding [38] (as used by multihashes [25]) is *not* used here; COER has its own way of encoding the lengths of octet strings and integers, and there is no need to duplicate that information.

In this version of circulex, implementations MUST support SHAKE256 [7] with 512 bits (64 bytes) of output.

3.5 Signature scheme

Although circulex allows future flexibility in the choice of signature scheme, only one signature scheme is specified for use in this version of circulex; it's inspired by *EdDSA for more curves* [4], but uses ristretto255 [39] as the underlying group.

Let $\ell = 2^{252} + 27742317777372353535851937790883648493$, the order of the ristretto255 group; let B denote its standard basepoint. The canonical encoding of an element A of the group is denoted $\underline{A}$. A scalar S for the group is an integer modulo ℓ, and is encoded as $\underline{S}$ using the usual 256-bit little-endian encoding of $\{0, 1, \ldots, \ell - 1\}$.

Let H denote SHAKE256 with 512 bits of output [7]; when its output is to be used as an integer, it is to be interpreted as the little-endian encoding of an integer in $\{0, 1, \ldots, 2^{512} - 1\}$. For integers i and j such that $0 \le i < j \le 512$, let $h_i(x)$ denote bit i of the output of $H(x)$ (the first bit being counted as bit 0) and $h_{i,j}(x)$ denote the part of the output of $H(x)$ from bit i to bit $j - 1$, inclusive, so that $h_{0,512}(x) = H(x)$, $h_{i,i+1}(x) = h_i(x)$, and $h_{i,j}(x) \| h_{j,k}(x) = h_{i,k}(x)$ for $0 \le i < j < k \le 512$. Let $s(x) = 2^{251} + \sum_{0 \le i < 251} 2^i h_i(x)$, so that $s(x)$ is a nonzero scalar.

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 calculates its corresponding public key $\underline{A} = \underline{s(k)B}$. Along with k, the scalar $s(k)$ MUST be kept secret. When the instance 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 $\left(\underline{A_0}, k_0\right)$, and instance 1 has medium-term key pair $\left(\underline{A_1}, k_1\right)$; each instance's medium-term public key is known to the other instance. Each instance MUST verify that the other's public key is not 0. Instance i calculates a group element $A_{i,1-i}$ 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 group element for instances 0 and 1, and they MUST keep it secret.

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 verification under $\underline{T}$. Instance 1 generates some data x and sends it to instance 0. Each instance calculates $T = s\left(\underline{A_{0,1}} \middle\| x\right) A_0$; only instance 0 knows a discrete logarithm of T base B, specifically $t = s\left(\underline{A_{0,1}} \middle\| x\right) s(k_0)$, which MUST be kept secret.

Given a message M on which instance 0 wishes to construct a signature that passes verification under the public key $\underline{T}$, it calculates $r = H(h_{256,512}(k_0) \| x \| M)$. (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 $S = (r + H\left(\underline{R} \| \underline{T} \| M\right) t) \mod \ell$ and outputs the signature $(\underline{R} \| \underline{S})$.

If an instance with key pair $(\underline{A}, k)$ wishes to construct a signature that passes verification under $\underline{A}$ (rather than under a public key generated for it by a peer), it proceeds as in the previous paragraph, but with $T = A$, $t = s(k)$, and x being empty, so that $r = H(h_{256,512}(k) \| M)$.

When an instance wishes to verify that a purported signature $(\underline{R}, \underline{S})$ on a message M is valid under a public key $\underline{T}$, it parses $\underline{R}$ and $\underline{T}$ as group elements R and T, respecively, and $\underline{S}$ as a scalar S. It then checks that $0 \leq S < \ell$ and $SB = R + H\left(\underline{R} \| \underline{T} \| M\right) T$. If the inequalities and the equality all hold, the signature is valid; otherwise it is invalid.

The warnings in *EdDSA for more curves* ([4], page 3) regarding different

choices of r SHOULD be heeded; implementers SHOULD NOT succumb to the temptation to create signatures using other choices of r, which may be accidentally repeated or too easily guessable, leading to an attacker discovering the signer's secret scalar and forging signatures.

The ASN.1 definitions for signature schemes generally are:

```
-- Signatures
SignatureScheme ::= CHOICE {
    experimental            Experimental,
    ristretto255-shake256   NULL,
    ...
}
```

Signature schemes MUST ensure that concatenated encoded keys can be uniquely parsed, thus recovering the original sequence of keys, and that concatenated signatures can likewise be uniquely parsed, recovering the original sequence of signatures. The signature scheme specified above achieves this by encoding keys and signatures as fixed-length sequences of bytes.

```
PublicKey ::= SEQUENCE {
    scheme  SignatureScheme,
    key     OCTET STRING
}
```

The elements of a `PublicKey` are:

`scheme`
: The signature scheme to be used with this public key.

`key`
: The public key itself.

The message M for which a signature is created MUST be the COER encoding of a `Signable` object.

```
Signable ::= CHOICE {
    experimental        Experimental,
    hintBody            HintBody,
    paymentPath         SignablePath,
    completeAgreement   SignablePath,
    ...
}
```

See below for the definitions of `HintBody` and `SignablePath`, as well as further conditions restricting the circumstances under which signatures may be created.

3.6 Communication with peers

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 libp2p [16] over QUIC [11]. The libp2p protocol id [26] for circulex is `/circulex/` followed by the protocol version number, which is compliant with Semantic Versioning 2.0.0 [30]. The version number for this version of the circulex protocol is `1.0.0-1674690536`.

```
-- Identifying and commmunicating with other participants
PeerID          ::= OCTET STRING
MultiAddress    ::= OCTET STRING
```

A `PeerID` is a libp2p peer ID [10], and a `MultiAddress` is a multiformats multiaddr [22]. In both cases, the binary encoding is used, rather than the string encoding.

Because the timely delivery of complete agreements is particularly important, instances SHOULD NOT send them over the same streams as messages whose punctuality is not so crucial; this policy can help avoid unnecessary delays due to head-of-line blocking in cases where parts of earlier, less important, messages have been lost and need to be resent.

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.

4 Definitions

4.1 Identity

An *identity* is used to specify a participant in circulex. It contains sufficient information to encrypt and authenticate connections to that participant, assuming a means of communicating with them has been established. This version of circulex has just one non-experimental alternative for identities — a libp2p peer ID.

```
Identity ::= CHOICE {
    experimental    Experimental,
    libp2p          PeerID,
    ...
}
```

4.2 Contact details

A *contact details* object is used to specify how instances can, for the time being, connect to a particular instance. Note that this information can change over time — for example, if the instance has a dynamic IP address and no domain name.

```
ContactDetails ::= SEQUENCE {
    identity    Identity,
    addresses   SEQUENCE OF MultiAddress
}
```

Where the `identity` is a libp2p peer ID, any `MultiAddress` listed in `addresses` MUST NOT end with the binary encoding of `/p2p/bafz...`, where `bafz...` represents the peer ID specified in `identity`; repetition of this information would be wasteful.

4.3 Currency

A *currency* is specified as follows:

```
-- Requesting payment (out-of-band)
Currency ::= CHOICE {
    experimental    Experimental,
    standard        IA5String
                        (SIZE (3) ^ FROM ("A".."Z")),
    userDefined     LimitedString,
    ...
}
```

The non-experimental alternatives have the following meanings:

`standard`
: The standard capitalized code representing the currency [35].

`userDefined`
: This alternative allows for the use on circulex of arbitrary obligations. Participants who make use of this feature to record obligations between each other MUST share a mutual understanding of the meaning of the obligations. Circulex implementations need not understand the meanings of user-defined obligations; however, they MUST ensure that the participants confirm that they have the required mutual understanding.

4.4 Amount

An *amount* specifies a certain quantity of a certain currency.

```
Amount ::= SEQUENCE {
    currency    Currency,
    numerator   INTEGER,
    denominator PositiveInteger
}
```

The `currency` element specifies the currency, while the `numerator` and `denominator` elements encode the numerator and denominator, respectively, of a fraction specifying the quantity of the currency.

4.5 Target

A *target* can contain information that might assist instances' pathfinding attempts for an intended transaction.

```
Target ::= SEQUENCE {
    currency    Currency OPTIONAL,
    payload     Amount OPTIONAL
}
```

Its elements have the following meanings:

`currency`
: A currency intended to be used in the transaction.

`payload`
: The approximate value intended to be transferred by each participant in the transaction. The currency specified in the payload SHOULD be one that will be widely understood, and can be the same as or different from the target's own currency (if present).

4.6 Reference header

A *reference header* is used to specify the interpretation of reference data to be included in a payment specification. Typically, the meaning of the header only needs to be understood by the payee's instance; the payer's instance will receive the header in a payment request and include it unaltered in a payment specification.

The ASN.1 definition of a reference header is:

```
ReferenceHeader ::= SEQUENCE {
    scheme      LimitedString,
    fixedField  LimitedString
}
```

The elements have the following meanings:

`scheme`
: A string identifying the reference scheme used by the payee for this payment. The payee doesn't need to ensure that no other payees are using the same string to identify a different scheme, but SHOULD NOT use the same string to identify different schemes in use by itself.

`fixedField`
: A fixed string to be included by the payer in a payment specification in a partial agreement sent to the payee. The payee can use this field for the payer's account number, or a code identifying a particular purchase, for example, but MAY leave it empty. The value of the `scheme` element MUST be sufficient to allow the payee to correctly interpret this field when it's included in a partial agreement sent to the payee.

4.7 Payment request

A *payment request* can be used to specify a request for a payment to be made via circulex.

```
PaymentRequest ::= SEQUENCE {
    payee           ContactDetails,
    amount          Amount,
    paywiseTarget   Target,
    referenceHeader ReferenceHeader,
    prompts         SEQUENCE OF LimitedString
}
```

Its elements have the following meanings:

`payee`
: The circulex participant to be paid, according to the request.

`amount`
: The currency and amount requested to be paid.

`paywiseTarget`
: The target requested to be used in paywise pathfinding for the transaction.

`referenceHeader`
: A reference header to be used verbatim as the header of the `reference` element of a payment specification that is to be included in a partial agreement between the payer (as the payee's paywise neighbour) and the payee, if the payment is to be made.

`prompts`
: A sequence of strings to be presented to the payer (if they are willing to make the payment). The payer's responses to those prompts are to be included (in the same order) in the `answers` element of the reference included in the partial agreement they send to the payee; the `answers` element MUST be a sequence of the same length as the `prompts` element of this reference template, even if some or all of the payer's responses are empty.

4.8 URI scheme

Some circulex-related information, such as that required to authenticate a participant, is transmitted out of band using Uniform Resource Identifiers [2].

A circulex URI begins with the string "`circulex:`", which is followed by a case-sensitve string identifying the type of data contained in the URI, then an exclamation mark ("`!`"), and finally a multibase encoding [23] of the COER encoding of the relevant data. Notice that the only delimiters used in a circulex URI are the single colon after the URI scheme name and the single exclamation mark after the data type indicator. In particular, the base chosen for the multibase encodings MUST NOT use an alphabet with any characters outside the set of unreserved characters for URIs.

This version of circulex defines three data type indicators, with their associated data types, as follows:

`x` An object of the `Experimental` type, whose use and interpretation is defined by the experiment.

`c` A contact details object, which can be used by a participant to give their potential peers the necessary information to make and authenticate a connection. This deliberately excludes any claim about the external identity of the owner of the contact details, so that an application processing such a URI needs to prompt its user for that information; this prevents scammers from producing `circulex:c!...` URIs that contain their own cryptographic identities, but claim to belong to a popular charity, for example, which might be relied on by a poorly designed circulex implementation.

`p` A payment request. Again, and for the same reason, this deliberately excludes claims about the external identity of the participant requesting the payment.

There are two RECOMMENDED forms of circulex URIs. The first is intended for use in text-based communications; this form uses unpadded base64url [13] in the URI, which is then encoded in UTF-8 [45]; readers SHOULD silently ignore any whitespace characters in such URIs that are presented to them. The second form uses the airtameg encoding [18] in the URI, which is then encoded via UTF-8 into an Aztec Code [17].

4.9 Path types

As it does with signature schemes, circulex allows future flexibility in the types of payment paths that are used, but defines only one path type for this version of circulex, the *simple path*.

```
-- Path types
SimplePath ::= SEQUENCE {
    talkwise    BOOLEAN,
    keys        OCTET STRING
}
```

`talkwise`
: If `TRUE`, then the keys in this path are listed in talkwise order, each key being the talkwise neighbour of the previous one; otherwise the keys are listed in the opposite—paywise—order.

`keys`
: The concatenation of the keys of the participants (or potential participants) in a transaction. The signature scheme and transaction to which the keys belong is determined by the context in which this simple path is included.

For path types generally, we have the following specifications:

```
PathType ::= ENUMERATED {
    experimental,
    simple,
    ...
}

PATHTYPES ::= CLASS {
    &pathType       PathType UNIQUE,
    &UnsignedPath
} WITH SYNTAX {&pathType, &UnsignedPath}

PathTypes PATHTYPES ::= {
    {experimental,  Experimental} |
    {simple,        SimplePath  },
    ...
}
```

4.10 Transaction identifier

A *transaction identifier* is used not only to uniquely identify a particular transaction, but also to specify the path type for the transaction.

```
TransactionIdentifier ::= SEQUENCE {
    completionDeadline  TimeLabel,
    finalityDeadline    TimeLabel,
    pathType            PathType,
    initiatorPublicKey  PublicKey
}
```

The elements have the following meanings:

`completionDeadline`
: 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.

`finalityDeadline`
: The *finality deadline* of the transaction — the time by which the disposition of the transaction will be finalized. Reversal deadlines in partial agreements for this transaction MUST NOT be later than the finality deadline.

`pathType`
: The path type to be used in this transaction.

`initiatorPublicKey`
: 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.

4.11 Signed payment path

A *signable path* identifies a specific path for a specific transaction.

```
SignablePath ::= SEQUENCE {
    transaction TransactionIdentifier,
    path        PATHTYPES.&UnsignedPath ({PathTypes})
}
```

The `path` member of a signable path MUST be of the `&UnsignedPath` type associated (in the `PathTypes` table) with the `pathType` member of the `transaction` member of the same signable path.

In the case of simple paths, the keys listed in the path MUST be public keys belonging to the same signature scheme as is specified for the `initiatorPublicKey` in the transaction identifier.

A *signed payment path* consists of a signable path together with signatures indicating the conditional agreement (to participate in the transaction) of the participants listed in the signable path.

```
SignedPath ::= SEQUENCE {
    signablePath    SignablePath,
    signatures      OCTET STRING
}
```

In the case of simple paths, the `signatures` member MUST be the concatenation of m signatures, where m is the number of keys in the `keys` member of the `path` member of the `signablePath` member of the signed payment path. Signature number j (where $1 \leq j \leq m$) MUST be valid under key number $j - 1$ (where key number 0 is the initiator public key), according to the signature scheme specified for the initiator public key. It MUST be a valid signature of the COER encoding of a `Signable` object using the `paymentPath` alternative, where the `SignablePath` has the same transaction identifier as that in the signed payment path, and where the `SignablePath` being signed contains the same keys in the same order as the `signablePath` member of the signed payment path, but excludes all of the keys after key number j.

Definitions of other path types MUST specify requirements for the contents of the `signatures` member of signed payment paths. These rules MUST be sufficient to establish and verify the agreement of all participants whose keys are listed in a valid complete agreement for a transaction that uses that path type.

It's RECOMMENDED that the signatures are required to be listed in the signed payment path in the same order as the keys under which they're valid are listed in the signable path. It's also RECOMMENDED that a canonical subpath (also of the type `SignablePath`) is defined, containing all of the predecessors and immediate successors of a public key listed in the path; the `signatures` member of a signed payment path SHOULD be the concatenation of signatures (valid under the signature scheme specified in the `initiatorPublicKey` in the transaction identifier of the path) of `Signable` objects using the `paymentPath` alternative, where the `SignablePath`s are canonical subpaths, as defined in accordance with this paragraph's recommendations.

Signatures of `Signable` objects using the `paymentPath` alternative MUST NOT be created except in accordance with the requirements specified in Section 5.11.3.

5 Messages

There are many different types of circulex message, and the details of each non-experimental message type are explained in the following subsections.

```
    -- Messages
    MessageType ::= ENUMERATED {
        experimental,
        invitation,
        statement,
        relayOffer,
        ping,
        pong,
        relayRequest,
        relayPeering,
        latencyReport,
        relayRejection,
        hint,
        partialAgreement,
        completeAgreement,
        receipt,
        tally,
        missingInformation,
        freeze,
        ...
    }

    SENDABLE ::= CLASS {
        &type       MessageType UNIQUE,
        &BodyType
    } WITH SYNTAX {
        &type,
        &BodyType
    }

    Sendable SENDABLE ::= {
        {experimental,          Experimental                } |
        {invitation,            Invitation                  } |
        {statement,             Statement                   } |
        {relayOffer,            RelayOffer                  } |
        {ping,                  Ping                        } |
```

```
        {pong,                  Pong                        } |
        {relayRequest,          RelayRequest                } |
        {relayPeering,          RelayRequestContext         } |
        {latencyReport,         LatencyReport               } |
        {relayRejection,        SEQUENCE OF Identity        } |
        {hint,                  Hint                        } |
        {partialAgreement,      PartialAgreement            } |
        {receipt,               Receipt                     } |
        {tally,                 Tally                       } |
        {missingInformation,    MissingInformation          } |
        {freeze,                TimeLabel                   },
        ...
    }

    Message ::= SEQUENCE {
        quoted  BOOLEAN,
        type    SENDABLE.&type ({Sendable}),
        body    SENDABLE.&BodyType ({Sendable}{@type})
    }
```

The members of a message have the following meanings:

`quoted`
: This flag MUST be set if, and only if, the purpose of this message is to send back to its original sender the contents of a message that was originally sent in the opposite direction to this one. This will typically be in response to certain kinds of missing information request.

`type`
: The type of this message.

`body`
: The body of the message.

5.1 Invitation

5.1.1 Bandwidth limit

When included in a message sent between circulex peers, a *bandwidth limit* conveys the sender's desired upper limit on the bandwidth used by a certain class of circulex messages; the particular class that it relates to is determined by context. The limit excludes the overheads associated with the relevant transport and encryption protocols.

```
-- Invitation
BandwidthLimit ::= SEQUENCE {
    bytes   NonnegativeInteger,
    seconds PositiveInteger
}
```

The members of a bandwidth limit have the following meanings:

`bytes`
: The desired maximum number of bytes used by the relevant class of messages over each period of the specified duration of time.

`seconds`
: A number of seconds, specifying the duration relevant to this bandwidth limit.

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.

Bandwidth limits are specified in sequences, allowing messages to specify multiple concurrent limits, each relating to a different duration. For example, a message might specify a limit of 60 MiB per day, with bursts not exceeding 400 kb/s, using the following sequence of bandwidth limits:

`01 02`	2 bandwidth limits:
	First bandwidth limit:
`04 03c00000`	60 MiB in each period of
`03 015180`	86,400 seconds;
	Second bandwidth limit:
`02 c350`	50,000 bytes in each
`01 01`	1 second.

A sequence of bandwidth limits SHOULD NOT include multiple limits specifying the same duration. In a sequence of bandwidth limits, limits specifying longer durations SHOULD imply lower average bandwidth usage than limits in the same sequence that specify shorter durations.

5.1.2 Invitation

An *invitation* is a message that can be used by an instance to inform a peer of its currently preferred hash algorithms, path types, medium-term public keys, bandwidth limits, aliases, and relays.

```
Invitation ::= SEQUENCE {
    hashAlgorithms  SEQUENCE OF HashAlgorithm,
    pathTypes
        SEQUENCE OF SEQUENCE {
            pathType        PathType,
            mediumTermKey   PublicKey
        },
    groupedBandwidth
        SEQUENCE OF SEQUENCE {
            addresses   SEQUENCE (SIZE (1 .. MAX)) OF
                MultiAddress,
            limits      SEQUENCE OF BandwidthLimit
        },
    aliases         SEQUENCE OF Identity,
    relays          SEQUENCE OF ContactDetails,
    relayBandwidth  SEQUENCE OF BandwidthLimit,
    sentAt          TimeLabel
}
```

Its members have the following meanings:

`hashAlgorithms`
: The hash algorithms that can be used for concisely and uniquely identifying this invitation in other messages sent between the two instances. If this sequence is empty, or otherwise lists no algorithms supported by the recipient, then this invitation can't easily be used as part of new bilateral agreements.

`pathTypes`
: The path types and associated medium-term public keys that the sender is willing to use for new agreements. This sequence MUST NOT include multiple elements with the same path type and signature scheme. In order for two instances to participate in transactions together, invitations sent in each direction between them need to agree on at least one path type and associated signature scheme.

`groupedBandwidth`

Groups of addresses the sender can be contacted on (again excluding any redundant trailing `/p2p/bafz...`) and any bandwidth limits associated with each group. The bandwidth limits in an item in the `groupedBandwidth` sequence don't apply individually to the addresses in the same item; they apply to the *total* bandwidth used by qualifying messages sent to *any* of those addresses.

The class of messages these limits relate to consists of the following messages sent by the recipient of the invitation to its sender:

- invitations,
- statements,
- relay offers,
- relay rejections,
- hints,
- partial agreements,
- missing information requests for any of the above, and
- freeze messages.

An empty sequence of bandwidth limits implies that the sender wishes to place no limitation on the bandwidth used by such messages when they're sent to any of the associated addresses.

`aliases`

Other identities controlled by the sender. These can be used, for example, to inform the recipient that the sender controls one or more identities from which it will soon send relay offers. Using different identities in its role as a relay might help to hide the network of trust relationships from any attempted authoritarian surveillance, but is unlikely to be effective if all the identities are contacted via the same addresses. Also, if a different alias is used for each peer, surveillance of those peers might become easier, rather than harder; on the other hand, surveillance might become particularly difficult if aliases and addresses are frequently changed.

Another possible use for aliases could be to facilitate a participant's transition to a new identity on a new device, without requiring human intervention from all their peers.

An instance MUST NOT rely on a participant's purported alias until it has received both an invitation from one of the participant's established

identities, listing the new alias, and an invitation from the new alias, listing at least one of the established identities.

`relays`
: 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. If this list has fewer than three elements, then this invitation can't be used to create new bilateral agreements.

`relayBandwidth`
: Bandwidth limits that apply to the relay peering messages, pings, and complete agreements sent by the recipient's relays to a typical one of the sender's relays.

`sentAt`
: The time at which this invitation was originally sent. This allows the recipient to identify the most recent invitation, even if invitations are reordered in transit or received multiple times. The recipient SHOULD verify that the value of this member encodes a point in time in the past. If the invitation is resent in response to a missing information request, the value of this member MUST NOT be updated.

When an instance wants to invite another instance to be its peer, it MUST send an invitation to that peer. 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 it wants to make changes to its preferred hash algorithms, path types, medium-term public keys, aliases, or relays. It SHOULD send new invitations whenever its desired bandwidth limits change significantly.

An instance MAY put quite different contents, including different sets of relays, into invitations it sends to different peers; however, participation with two of its peers in the same circular exchange requires it to share the same path type and signature scheme with both of those peers; there are also requirements relating to the overlap between sets of relays that are to be used in the same transaction. An instance 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.

5.2 Statement

5.2.1 Statement subject

A *statement subject* expresses the context for a statement.

```
-- Statement
StatementSubject ::= SEQUENCE {
    fromTime    TimeLabel,
    toTime      TimeLabel,
    currency    Currency
}
```

Its elements have the following meanings:

`fromTime`
: The start of the period of time the statement relates to.

`toTime`
: The end of the period of time the statement relates to. This MUST NOT be earlier than the start of the period of time.

`currency`
: The currency the statement relates to.

5.2.2 Range

A *range* represents a range of possible integers.

```
Range ::= SEQUENCE {
    minimum INTEGER OPTIONAL,
    maximum INTEGER OPTIONAL
}
```

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).

5.2.3 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 themself and the recipient.

```
    Statement ::= SEQUENCE {
        subject             StatementSubject,
        denominator         PositiveInteger,
        openingBalance      INTEGER,
        closingBalance      INTEGER,
        agreements          SEQUENCE OF PublicKey,
        balanceLimits       Range,
        transactionSizes    Range,
        maximumOutstanding  NonnegativeInteger OPTIONAL,
        durationOutstanding NonnegativeInteger OPTIONAL,
        sentAt              TimeLabel
    }
```

`subject`
: The subject of this statement, specifying the currency and period of time it relates to.

`denominator`
: Most of the remaining members of this statement, described below, specify amounts or ranges in the currency specified in the `subject`. In each case, a quantity in the currency is specified by a fraction in which the denominator is the value of this `denominator`; the numerator is an integer 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 sender to the recipient; a negative number represents an obligation in the opposite direction. This denominator is also used to indicate which denominators the sender would find acceptable in payment specifications for the relevant currency. Therefore, the sender SHOULD use the largest acceptible denominator here.

`openingBalance`
: A numerator indicating the outstanding obligation at the start of the period specified in the subject of this statement, taking into account all relevant bilateral agreements whose reversal deadlines are strictly earlier than the start of that period.

`closingBalance`
: A numerator indicating the outstanding obligation at the end of the period specified in the subject of this statement, taking into account all relevant bilateral agreements whose reversal deadlines are strictly earlier than the end of that period.

`agreements`
: A list of all the 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. Each such bilateral agreement is identified by the public key associated with it, generated as described in Section 5.11.3.

`balanceLimits`
: A range indicating the sender's current intentions (when the message is sent) 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.

`maximumOutstanding`
: A numerator 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.

`durationOutstanding`
: The maximum number of nanoseconds the sender intends to allow any one transaction to remain of uncertain disposition.

`sentAt`
: The time at which the sender sent this statement.

The `balanceLimits`, `transactionSizes`, `maximumOutstanding`, and `durationOutstanding` 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 between

the sender and recipient; they are merely indicative of which bilateral agreements the sender is likely to agree to.

If an instance wants to establish a relationship (in a particular currency) with another instance (for example, extending credit in that currency to the other instance), 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, it SHOULD respond with its own statement whose subject specifies the same currency as in the statement it received.

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 to the recipient, (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`, `maximumOutstanding`, or `durationOutstanding` members 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. 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 with 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`.

5.3 Relay offer

5.3.1 Latency probability object

A *latency probability object* is defined as follows:

```
-- Relay offer
LatencyProbability ::= SEQUENCE {
    latency     INTEGER,
    probability REAL (0 .. 1)
}
```

`latency`
: A number of nanoseconds.

`probability`
: A probability.

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 specific expiry time, complete a particular task within at most the specified number of nanoseconds, with at least the specified probability. The specific expiry time and task are determined by the context of the message in which the latency probability object is included.

Latency probability objects aren't intended to be used in isolation; instead, a sequence of latency probability objects is typically used. The sender of such a sequence 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.

A sequence of latency probability objects SHOULD NOT include any element whose information is logically implied by any other included element. For example, if one element states that a latency of no more than 120 ms is achievable with a probability of 0.9, then another element that states that a 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.

5.3.2 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.

```
RelayOffer ::= SEQUENCE {
    pathTypes
        SEQUENCE (SIZE (1 .. MAX)) OF SEQUENCE {
            pathType        PathType,
            signatureScheme SignatureScheme,
            latencies       SEQUENCE OF LatencyProbability
        },
    validUntil          TimeLabel,
    addresses
        SEQUENCE (SIZE (1 .. MAX)) OF MultiAddress,
    bandwidthLimits     SEQUENCE OF BandwidthLimit,
    p2pCircuitRelay     BOOLEAN
}
```

`pathTypes`
: The path types and associated signature schemes that the sender will generally find acceptable for transactions covered by relay requests sent by the recipient of this relay offer, together with information about the time the relay is likely to spend processing complete agreements of each type.

  For the purpose of the latency probability objects, the relevant duration begins at the time the relay has finished receiving a complete agreement of the specified type, and ends when the relay has finished recognizing and validating the complete agreement, and has determined which destinations it's obliged to forward it to. If the sender is going to act as a relay for the recipient using multiple associated aliases, then the latency measurements MUST take into account the time required to disseminate to all such aliases (if they're running on different machines) a complete agreement received by the identity from which this offer is sent. The relevant expiry time is determined by the value of the `validUntil` member, below.

`validUntil`
: A time label, indicating that the sender of this relay offer will generally assent to transaction relay requests from its recipient if their transactions' finality deadlines are no later than this time.

`addresses`
: The addresses the sender is willing to use in its capacity as a relay. Unless another message authorizes it, the recipient MUST NOT send any relay-specific messages to any other addresses for this relay, or include other addresses for this relay in the `relays` member of invitations it sends to its peers. These addresses are associated with the identity from which this offer is sent. If that identity is a libp2p peer ID, then the addresses MUST NOT include a redundant trailing `/p2p/bafz...`.

`bandwidthLimits`
: A list of bandwidth limits. The class of messages these limits relate 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 relay peering messages, pings, and complete agreements sent to the sender by any of the recipient's peers' relays.

`p2pCircuitRelay`
: A flag indicating whether the sender is also willing to act as the recipient's libp2p circuit relay [29], as well as acting as its circulex relay. When an instance connects through a circuit relay, that instance and the recipient SHOULD attempt to replace their connection with a direct connection as quickly as possible.

5.4 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.

```
-- Ping
Ping ::= SEQUENCE {
    sentAt  TimeLabel,
    padding OCTET STRING
}
```

`sentAt`
: A time label encoding the time at which the ping was sent.

`padding`
: 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. However, instances and relays MUST NOT send large or numerous pings to destinations from which they haven't received a current relay offer or relay peering message.

A relay MUST, in general, reply to each received ping with a corresponding pong, as quickly as possible, assuming it has already sent a current relay offer or relay peering message to the instance that sent the ping. However, it SHOULD implement sufficient rate-limiting to protect itself 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.

5.5 Pong

A *pong* is a message sent in response to a ping.

```
-- Pong
Pong ::= SEQUENCE {
    pingSentAt  TimeLabel,
    receivedAt  TimeLabel
}
```

`pingSentAt`
: The time label included in the ping to which this pong is a reply.

`receivedAt`
: 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 the 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.

5.6 Relay request

5.6.1 Relay request context

A *relay request context* is a component of a relay request that indicates whether the request is an indicative one, for a specific length of time, or whether it relates to a specific transaction.

```
-- Relay request
RelayRequestContext ::= CHOICE {
    until       TimeLabel,
    transaction TransactionIdentifier
}
```

`until`
: This alternative is used in indicative relay requests. The time specifies the end of the period that the request relates to.

`transaction`
: This alternative is used in transaction relay requests. The request relates to the specified transaction.

5.6.2 Relay request

A *relay request* is a message that can be used by an instance to request a latency report from one of its desired relays.

```
RelayRequest ::= SEQUENCE {
    context         RelayRequestContext,
    destinations    SEQUENCE OF ContactDetails
}
```

`context`
: The context of this relay request.

`destinations`
: The list of destinations to include in the requested latency report.

There are two kinds of relay requests: indicative relay requests and transaction relay requests.

Indicative relay request Suppose an instance has received a relay offer from a peer. Before relying on that relay for specific transactions, 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 an *indicative relay request*.

The recipient of an indicative relay request SHOULD respond to it with a latency report if it would generally assent to transaction relay requests from the same sender in which:

- the finality deadline for the transaction is no later than the time label in the context of the indicative request, and
- the list of destinations is equal to or a subset of the list in the indicative relay request

(provided that the transaction is of a type that a current relay offer indicated was acceptable); otherwise it MUST NOT send a corresponding latency report.

Transaction relay request A *transaction relay request* is used to request not only a latency report, but also a commitment to act as a relay for a specific transaction. The recipient of such a request MUST respond with a corresponding latency report if, and only if, it assents to the 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 SHOULD NOT include the same destinations in a later relay request for the same transaction.

Once a relay has agreed to a transaction 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 between the transaction's completion and finality deadlines (inclusive of the completion deadline, but exclusive of the finality deadline), 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 5.11.2 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.

Where possible, instances SHOULD try to reduce the risk of head-of-line blocking unnecessarily delaying the transmission of complete agreements. For example, where QUIC is being used, each complete agreement SHOULD be sent in a stream of its own. Where QUIC stream prioritization is available, complete agreement streams SHOULD be given the highest priority.

A relay MUST NOT agree to a transaction 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 SHOULD NOT forward it again to destinations that have already received it; also a relay SHOULD NOT send a complete agreement back to a destination from which it knows it has received that complete agreement, even if it received it from that destination before the completion deadline.

A relay that has agreed to a transaction 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 and finality deadlines.

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 finality deadline of the 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 a receipt.

5.7 Relay peering message

A *relay peering message* is used by a relay to indicate its intention to relay complete agreements to, and accept them from, the recipient of the message. Its body is a `RelayRequestContext`.

If its body uses the `until` alternative, it's an *indicative relay peering message*. If a relay intends to send a latency report corresponding to an indicative relay request, it MUST ensure that it has sent to each included destination an indicative relay peering message covering the relevant period of time.

If a relay peering message's body uses the `transaction` alternative, it's a *transaction relay peering message*. If a relay intends to send a latency report corresponding to a transaction relay request, it MUST first send to each included destination a transaction relay peering message for that transaction. Before including the destination in the latency report, it MUST have received from that destination a transaction relay peering message for that transaction.

5.8 Latency report

5.8.1 Latency profile

A *latency profile* is defined as follows:

```
-- Latency report
LatencyProfile ::= SEQUENCE {
    destination Identity,
    profile     SEQUENCE OF LatencyProbability
}
```

`destination`
: The destination this profile relates to.

`profile`
: Information about latencies to the destination that are achievable with various probabilities. The context in which the latency profile is included determines the relevant expiry time for the latency probability objects.

The task the latency probability objects relate to is the sending of a complete agreement to the specified destination. The relevant measurement is from the time (according to the relay's clock) at which the relay has determined that it's obliged to forward the complete agreement to the destination, 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 of an included latency probability object 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 latency profile 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.

5.8.2 Latency report

A *latency report* is a message sent in reply to a relay request, indicating assent to the request.

```
LatencyReport ::= SEQUENCE {
    requestContext  RelayRequestContext,
    latencyProfiles SEQUENCE OF LatencyProfile
}
```

`requestContext`
: The relay request context included in the relay request to which this latency report is a reply.

`latencyProfiles`
: For each of these latency profiles, its `destination` member is the `identity` member of an element of the `destinations` member of the original relay request. The sender of this latency report SHOULD include a latency profile for each destination in that request, except for any destinations that it's currently unable to communicate with. In the case of an indicative relay request, the expiry time for these latency profiles is the value of the `until` alternative of the relay request context. In the case of a transaction relay request, the transaction in question is identified by the transaction identifier included in the `transaction` alternative of the relay request context, and the expiry time for the latency profiles is the finality deadline of that transaction.

5.9 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*.

The body of a relay rejection is a sequence of 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.

5.10 Hint

5.10.1 Hint subject

A *hint subject* specifies the transaction a hint relates to, as well as the direction the hint is intended to travel in.

```
-- Hint
HintSubject ::= SEQUENCE {
    talkwise    BOOLEAN,
    transaction TransactionIdentifier
}
```

`talkwise`
: If this flag is set, it indicates that the hint is a *talkwise hint*, to be sent by instances to their potential talkwise neighbours for the transaction; otherwise it's a *paywise hint*, to be sent by instances to their potential paywise neighbours.

`transaction`
: The transaction the hint relates to.

5.10.2 Hint body

A *hint body* specifies the subject and target of a hint.

```
HintBody ::= SEQUENCE {
    subject     HintSubject,
    target      Target
}
```

5.10.3 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.

```
Hint ::= SEQUENCE {
    body        HintBody,
    signature   OCTET STRING
}
```

`body`
: The body of the hint.

`signature`
: A valid signature, under the initiator's public key, of the COER encoding of a `Signable` object that uses the `hintBody` alternative and contains the value of the `body` member of this hint.

The initiator of a transaction MUST NOT create a signature that is valid 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.

If a talkwise hint's `currency` member is present, it SHOULD specify a currency the initiator expects to be used to pay its paywise neighbour or another participant not far from it in the paywise direction. Similarly, if a paywise hint specifies a currency, it SHOULD be one the initiator expects to be used in the payment made by its talkwise neighbour or another participant not far from it in the talkwise 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
- 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.

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 with an empty target (that is, one that doesn't specify a particular currency or payload); 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.

5.11 Partial agreement

5.11.1 Payment specification

A *payment specification* is used in a message sent between instances to specify the details of a proposed payment between those instances.

```
-- Partial agreement
PaymentSpecification ::= SEQUENCE {
    amount      Amount,
    reference
        SEQUENCE {
            header  ReferenceHeader,
```

```
            answers SEQUENCE OF LimitedString
        } OPTIONAL
    }
```

`amount`
: The amount of the payment. Its `denominator` SHOULD be a common factor of the denominators specified in the most recent statements (for the relevant currency) sent in each direction between the sender and recipient of the message containing this payment specification.

`reference`
: If this element is absent, it 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.

  If this element is present, it 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 instances owned by the same person. When the payment specification is based on a prior payment request, the `header` element is copied directly from the `referenceHeader` element of the payment request. The `answers` are the paywise partner's responses to the `prompts` in the payment request. In general, because this requires information from the payer, a payment specification that contains a reference will be first sent in the talkwise direction from the payer to the payee, but once the payee has the information, they can include it in paywise partial agreements, thus broadening the options for successful pathfinding. Alternatively, the talkwise partner can send the first payment specification with a reference if they're using a reference scheme that requires no information from the paywise partner; in this case, an empty sequence is used as the `answers` element.

Example 2. 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 creates a payment request using his reference scheme "v1" and containing the order reference "441a25a"; he requires no further information from Alice, as she supplied her shipping address when she placed 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:

```
80                          The reference element is present.
                            Amount:
  81 4b5257                   Standard currency: KRW
  03 06ddd0                   Numerator: 450,000
  01 01                       Denominator: 1
                            Reference:
                              Header:
    02 c681                     Scheme: "v1"
    07 848481b18285b1           Fixed field: "441a25a"
  00                          Zero answers
```

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 3. Gareth is a gardener who pays for goods and services via circulex by promising to do gardening work for his friends and relatives. He uses circulex to record (among other things) the number of hours of gardening work he owes each of his friends; Hermione is one such friend. This payment specification is for 1 hour and 20 minutes of work to be done in Hermione's garden (assuming it's used as part of the basis of a successful transaction, having been sent between Gareth's and Hermione's instances, with Gareth's being the talkwise partner):

```
00                          The reference element is absent.
                            Amount:
  82 18                       User-defined currency in 24 bytes
    98bfc5c2c370bfb67097b1c2    "Hours of Gareth's labour"
    b5c4b877c370bcb1b2bfc5c2
  01 50                       Numerator: 80
  01 3c                       Denominator: 60
```

5.11.2 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.

```
BilateralAgreement ::= SEQUENCE {
    deadline            TimeLabel,
    talkwiseInvitation  Hash,
    reversalDeadline    TimeLabel,
    paywiseInvitation   Hash,
    payments
        SEQUENCE (SIZE (1 .. MAX)) OF PaymentSpecification
}
```

`deadline`
: This agreement's deadline.

`talkwiseInvitation`
: The hash of the COER encoding of an invitation 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.

`reversalDeadline`
: This agreement's reversal deadline, which MUST be later than the deadline specified above.

`paywiseInvitation`
: The hash of the COER encoding of an invitation 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.

`payments`
: One or more payments to be made by the talkwise partner for the benefit of the paywise partner. Multiple payment specifications in the sequence specify components of a composite payment; they are *not* options from which one of the partners may choose one.

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.

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 relays aren't obliged to accept connections and carefully examine messages received from arbitrary sources, an instance MUST NOT assent to a bilateral agreement unless it's received satisfactory latency reports (corresponding to transaction relay requests for the relevant transaction) from all of its relays regarding all of its partner's relays.

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 (as determined by their `sentAt` members), and that they are hashed using an algorithm included in the `hashAlgorithms` members of both those invitations. Each hash MUST be of an invitation that listed at least three relays.

Example 4. 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.

`40000000573522aa3a7b2cd4`	Deadline: 2016–05–13 12:41:04.981150932
	Talkwise invitation:
`01 19`	SHAKE256
`40`	512-bit hash:
`7be3579ba7231c35`	
`f2d257ea4e64973f`	
`8a593afd3860b6c9`	
`a205d53ce736ee6d`	
`539a1c5f69904540`	
`f56217685392a544`	
`081ce12d2b35031a`	
`70690592ab95e31c`	
`40000000573522c22a8e8aec`	Reversal deadline: 2016–05–13 12:41:28.713984748
	Paywise invitation:
`01 19`	SHAKE256
`40`	512-bit hash:
`64d340cdf78ebf92`	
`489fbdbcb698a2a7`	
`96ffc58d5b35e375`	
`7c7e686d67cad606`	
`3652ae05f7756697`	
`eda1305479d567ba`	
`ab37480854c04322`	
`4be707fd08dc8746`	
`01 02`	Two payment specifications:
	First payment specification:
`00`	The `reference` element is absent.
	Amount:
`81 555344`	Standard currency: USD
`02 2710`	Numerator: 10,000
`01 64`	Denominator: 100
	Second payment specification:
`00`	The `reference` element is absent.
	Amount:
`81 4e5a44`	Standard currency: NZD
`03 00ea60`	Numerator: 60,000

`01 64` Denominator: 100

5.11.3 Partial agreement

A *partial agreement* is a message that can be used to incrementally construct a complete agreement for a transaction.

```
PartialAgreement ::= SEQUENCE {
    agreement   BilateralAgreement,
    path        SignedPath
}
```

`agreement`
: A bilateral agreement whose deadline is after the completion deadline of the transaction specified in the signed payment path below, and whose reversal deadline is no later than the finality deadline of that transaction.

`path`
: A signed payment path.

When a partial agreement is sent by an instance other than the initiator of the transaction, it MUST be the case that the sets of the sender's relays for the purposes of its talkwise and paywise bilateral agreements overlap in such a way that they each include more than half of the other set. One of these sets is specified by this partial agreement, and the other is specified by a partial agreement previously received by the sender of this one. Greater overlap adds to safety, but smaller overlap might make authoritarian surveillance of the sender more difficult. Another way to achieve such obfuscation would be to use the same set of relays, but ensure that they are identified by different peer IDs and contacted via different IP addresses.

In the case of a simple path with the ristretto255-shake256 signature scheme, where a partial agreement is the body of a message sent between instances, 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 role of the data x is played by the COER encoding of a `PartialAgreement` with the same contents as the one being sent, but omitting the last public key and signature, since they are yet to be determined when the key is being generated. Other signature schemes, if they are to be used with the simple path type, MUST specify rules that ensure that the last public key in the signed payment path is unique

to the bilateral agreement and all preceding parts of the signable path, and that only the recipient is able to create valid signatures under that public key. This last public key is the one used to identify this partial agreement for the purposes of statements and missing information requests.

In the case of the simple path type, a partial agreement can be created by the initiator of a transaction, or adapted by a recipient, who creates a new partial agreement by constructing a bilateral agreement they are willing to offer to one of their peers, adding the peer's temporary public key (generated in accordance with the rules 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 simple 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 in accordance with the rules above, and
  - the instance agrees to the bilateral agreement that was used to generate the key.

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 in accordance with the rules above,
  - the instance agrees to the bilateral agreement that was used to generate the key, 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).

Path types other than the simple path type MUST specify rules that similarly ensure that a valid complete agreement can only be created with the consent of all of the parties included in the agreement, that they can predict and limit which of their bilateral agreements can appear together in the same valid complete agreement, and that partial agreements can be uniquely identified by public keys for the purposes of statements and missing information requests.

5.12 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.

```
-- Complete agreements
CompleteAgreement ::= SEQUENCE {
    path            SignedPath,
    finalSignature  OCTET STRING
}
```

`path`
: A valid signed payment path. In the case of the simple path type, the `SimplePath` object it contains MUST list exactly 32 public keys, and the last public key MUST be the same as the initiator's public key. Other path types SHOULD ensure that the complete agreement leaks as little information as possible about who is or isn't included in the path, as the simple path type does here by fixing the size and structure of complete agreements, regardless of the actual number of participants in the path.

`finalSignature`
: A valid signature, under the initiator's public key, of the COER en-

coding of a `Signable` object using the `completeAgreement` alternative and containing the same `SignablePath` as the `SignedPath` above.

Two complete agreements are considered to be the same as each other if they are both valid and contain the same `SignablePath`, even if they disagree on 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 5.12.1 (though also being able to produce them in accordance with the rules of Section 5.12.3 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.

The rules in the subsections that follow are designed for path types in which every point in a path has exactly one immediate paywise neighbour, and exactly one immediate talkwise neighbour, which is the case for the simple path type. If another path type allows points in its paths to have multiple talkwise or paywise neighbours, then it MUST similarly specify rules that, if followed, prevent an initiator from accidentally incurring uncompensated or insufficiently compensated obligations; the rules SHOULD NOT prohibit potentially desirable uses of complete agreements, even if such uses would be unusual.

5.12.1 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.

5.12.2 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.

5.12.3 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.

5.12.4 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, in the simple path type, if the `talkwise` flag in the `SimplePath` object is unset, 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 5.12.1.

5.12.5 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 5.12.2'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 5.12.2, 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.

5.13 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.

```
-- Receipt
Receipt ::= SEQUENCE {
    receivedAt  TimeLabel,
    agreement   CompleteAgreement
}
```

`receivedAt`
: The earliest time at which the relay sending this receipt received the included 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).

`agreement`
: A complete agreement.

Because a receipt contains a whole complete agreement, an instance is considered to have received that complete agreement as soon as it has received a receipt that contains it.

5.14 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.

```
-- Tally
Tally ::= SEQUENCE {
    transaction     TransactionIdentifier,
    receiptsSent    INTEGER (0 .. 3)
}
```

`transaction`
: A transaction identifier.

`receiptsSent`
: 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 transaction's finality deadline. 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.

5.15 Missing information request

A *missing information request* is a message that can be used to request information that the sender doesn't have.

```
-- Missing information request
REQUESTABLE ::= CLASS {
    &type           MessageType UNIQUE,
    &IdentifiedBy
} WITH SYNTAX {
    &type,
    &IdentifiedBy
```

```
    }

    Requestable REQUESTABLE ::= {
        {experimental,      Experimental            } |
        {invitation,        Hash                    } |
        {statement,         StatementSubject        } |
        {hint,              HintSubject             } |
        {partialAgreement,  PublicKey               } |
        {receipt,           TransactionIdentifier   } |
        {tally,             TransactionIdentifier   },
        ...
    }

    MissingInformation ::= SEQUENCE {
        type            REQUESTABLE.&type ({Requestable}),
        identifiedBy
            REQUESTABLE.&IdentifiedBy ({Requestable}{@type})
    }

END
```

`type`
: The type of message being requested.

`identifiedBy`
: An object specifying which information of the relevant type is being requested. The following subsections explain the meaning of this member for each requestable type.

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.

Implementations MUST take care in determining when to respond to missing information requests with quoted messages, and when to respond with unquoted messages.

5.15.1 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 missing information request's `type` is `invitation`, and its `identifiedBy` member is the hash for which the corresponding invitation is being requested.

If an instance has lost its copy of one of its own invitations that it sent in the past (or never associated the invitation with its hash under the algorithm being used in a bilateral agreement), 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.

5.15.2 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 missing information request's `type` is `statement` and its `identifiedBy` member is a `StatementSubject` specifying the currency and period of time that statements are being requested for. The end of the period of time MUST NOT be in the future when the request is sent.

A valid response to such a request consists of one or more statements for the specified currency that together cover the specified period of time.

5.15.3 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 whose `type` is `hint` and whose `identifiedBy` member is the subject of the hint it wants to receive.

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.

5.15.4 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 `type` of the missing information request is `partialAgreement`, and its `identifiedBy` member is the public key whose associated partial agreement is being requested.

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

5.15.5 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 `type` of such a missing information request is `receipt` and its `identifiedBy` member is the transaction identifier of the transaction whose receipts are being requested.

The recipient of such a request SHOULD respond by resending the receipts that it sent to the requester for that transaction.

5.15.6 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 transaction's finality deadline, 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 `type` of the missing information request is `tally` and its `identifiedBy` member is 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 for the specified transaction, but only if it sent at least one latency report for the transaction to the requester, and the transaction's finality deadline has already passed.

5.16 Freeze message

A *freeze message* can be used by an instance whose secrets have, or might have, been leaked, to prevent further misuse of those secrets. Until the time specified in the body of the freeze message, the sender and recipient SHOULD refrain from sending unnecessary messages to each other, and MUST refrain from taking any actions that will or might result in any bilateral agreement between them becoming part of the basis of any new complete agreement. However, they MUST both continue to fulfil any commitments they've already made to each other, such as to relay complete agreements for certain transactions.

User interfaces SHOULD make it easy for their users to send simultaneous freeze messages to all their peers, initially with a body specifying a time in the near future (perhaps a few minutes away). The interface SHOULD then assist the user to understand the purpose and effect of freeze messages, so that the user can decide whether to send further freeze messages with a body specifying a time in the more distant future, giving them time to investigate whether their secrets have, in fact, been leaked, and whether to send final freeze messages with a body specifying a time so far in the future that the messages have essentially permanent effect.

If a user's peer has sent a permanent freeze message, the user will probably want to transfer their circulex relationship with that peer to the peer's new cryptographic identity. There are other reasons a user might want to perform such a transfer — for example, if their peer has lost the mobile phone that held their circulex secrets — so user interfaces SHOULD make it easy for users to perform such transfers (with the cooperation of the relevant peer), regardless of the reason, and regardless of whether the peer still has access to their old cryptographic identity.

6 Security considerations

Because circulex is intended to be private by design, and because its security depends in part on the privacy of certain secrets, it's useful to consider its security and privacy simultaneously. Furthermore, because it's intended as a decentralized system, this section will consider threats to "security" in the broadest sense, including threats to the decentralized nature of the network.

It is, however, worth considering different kinds of threats from different kinds of people or organizations. To put names to the threats, we have:

- OSIRIS, a terrorist organization that wishes to steal money from the circulex network or to disrupt financial infrastructure such as circulex,

but has no special technological capabilities;

- Big Brother [28], which wishes to extend its mass surveillance of communications to conduct mass surveillance of circulex activity; and
- Goliath Corporation [8], a large commercial organization, which wants to establish or maintain a market dominance in transactions sufficient to extract economic rents from them.

6.1 OSIRIS

6.1.1 Theft

Circulex's cryptography is designed to prevent OSIRIS from stealing money by forging signatures or otherwise impersonating other participants. Freeze messages are intended to limit what OSIRIS can do if it's discovered that it has stolen circulex secrets without stealing any physical device.

If a suitable alias has been set up on another device before the theft of the primary device, freeze messages could also be used to halt theft of funds even after the theft of a device containing circulex secrets. The device that holds the alias would need to be kept up to date about which destinations would need to receive the freeze messages in such an event. This defence could be integrated with more general-purpose privacy and security software that can be used to remotely disable a stolen device.

However, like any other participant, OSIRIS could easily steal money from anyone who chooses to trust it, by simply refusing to pay the debts it accrues through circulex.

A more difficult question to answer is whether OSIRIS can steal money by interfering with the timely transmission of complete agreements. To achieve this, OSIRIS would need to ensure a complete agreement reached its talkwise neighbour's relays on time (so that OSIRIS receives a payment from that neighbour), but prevent the complete agreement from reaching its own relays in time to trigger its obligation to pay its paywise neighbour. In order to have the complete agreement to send to its talkwise neighbour's relays, without having received it from its paywise neighbour's relays, OSIRIS would need to be the initiator of the transaction, but it could disguise this fact, even from its neighbours, by padding the partial and complete agreements it generates with extra public keys, and adding corresponding delays to deadlines and transmission of messages.

OSIRIS can easily prevent the complete agreement from reaching its own relays by ensuring that all of its relays are under its control, and taking them offline at the crucial time. However, this attack cheats only OSIRIS's

paywise neighbour, who has chosen to trust OSIRIS; circulex makes no claim to protect against such attacks. (If the trust relationship is one-way, and OSIRIS's payment was to be a reduction in the amount its paywise neighbour owed to it, then that neighbour can simply refuse to pay more than they think they owe OSIRIS.) It's worth noting, however, that unlike simply refusing to pay its debts, this attack can be made to appear not to be the deliberate choice of the attacker.

In a more sophisticated attack, OSIRIS might try to ensure that its paywise neighbour's relays are all offline at the crucial time, and therefore unable to forward the complete agreement to OSIRIS's relays. It might attempt to achieve this via a distributed denial-of-service attack on its paywise neighbour's relays' IP addresses, for example. If successful, this attack would result in OSIRIS's paywise neighbour's paywise neighbour (say, Bob) being obliged to pay his paywise neighbour, but not receiving a payment from OSIRIS's paywise neighbour (say, Alice), since her relays didn't receive the complete agreement on time; OSIRIS would receive a payment from its talkwise neighbour, but wouldn't make a payment to Alice.

Although this attack disadvantages Bob, who didn't explicitly choose to trust OSIRIS, the attacker, it's still the case that his trust in Alice was misplaced, as she was unable to keep her relays online when they were needed, due to her misjudgement about OSIRIS's trustworthiness. Although this doesn't expose circulex participants to attacks from people who are arbitrarily distant in payment chains (which would be a severe flaw in circulex), it does highlight that when choosing to trust someone, a circulex participant is also, to a certain extent, choosing to trust their judgement. (This would be true even in the absence of attacks like this one; by extending credit to someone, a participant is trusting that they won't get into unsustainable debts to other people, and that they won't unduly rely on debts owed to them by untrustworthy people.) Therefore, implementations MUST make this clear to users.

OSIRIS could also attempt to block the complete agreement at a point more distant from itself in the circular exchange, but it would need to know which relays to attempt to take offline, and it would need to obtain this information from the victim or one of the victim's neighbours. One way of achieving this would be to act as one of the relays for the victim or one of its neighbours, but in any case, if the attack is successful, it will be because the victim has trusted either OSIRIS itself or someone who shares information with OSIRIS, and because the victim, or someone they trust, was unable to keep their relays online at the crucial time. (This attack is also less likely to succeed than the previous one, since the complete agreement will be circulated not only by the relays of the participants in the chosen path,

but also by all of the relays of all of the *potential* participants, so there's a good chance that the complete agreement will find its way around the hole OSIRIS has tried to punch in the network of relays, and therefore that OSIRIS's paywise neighbour will be expecting payment, according to the relevant bilateral agreement.)

OSIRIS might also attempt attacks symmetrical to the ones described above, by interfering with the progress of a second complete agreement for a transaction, so that it reverses OSIRIS's obligation to pay its paywise neighbour, but fails to reverse OSIRIS's talkwise neighbour's obligation to pay OSIRIS. Such attacks give rise to the same considerations as above, and therefore no further requirements are necessary to prevent them.

6.1.2 Denial of service

Another of OSIRIS's potential goals is the disruption of financial infrastructure, including circulex. Using generic denial-of-service attacks against participants' IP addresses might succeed in forcing those participants offline during the attack, but circulex's decentralized nature limits the effects (on the network as a whole) of such attacks.

To try to affect the wider circulex network, OSIRIS might try making attractive but insincere promises to take part in transactions, thus wasting the resources of sincere would-be participants in the transactions. Circulex's design prevents OSIRIS from promising attractive exchange rates (for example) to transaction initiators without simultaneously making a firm commitment to its neighbours, conditional on that path being chosen.

However, OSIRIS can still, as a transaction initiator, make insincere pathfinding requests in an attempt to flood the network with partial agreements that it has no intention of completing. This could cause congestion on the network in a number of ways, such as the saturation of bandwidth limits or participants' `maximumOutstanding` limits. To prevent such an attack from reaching the wider network, implementations MUST ensure that each participant gives priority to partial agreements that are more likely to result in circular exchanges involving that participant, based on the characteristics of past successful and unsuccessful partial agreements, including at least the participant's immediate predecessors in those partial agreements. This will limit the effects of network flooding so that it primarily disadvantages only the attacker and those who are propagating the attack.

6.2 Big Brother

Big Brother is assumed to be a *passive pervasive attacker*, as defined in RFC 7624 [1]; we will also use that document's definition of *collaborator*, which includes unwitting collaborators. Big Brother wishes to use its surveillance capacity to discover as much as possible about circulex activity — who initiated which transactions, the value and purposes of those transactions, who trusts who, and so on.

If a participant's own device is a collaborator, there's very little that any system can do to prevent Big Brother from obtaining a great deal of information about them, except to ensure that as little historical information as possible is stored on the device when it's first compromised. Therefore, implementations SHOULD give users the option of automatic deletion of historical information that's been reconciled with the relevant peers' views on that information, and is therefore no longer required.

If a participant's peer is a collaborator, Big Brother will be able to access a lot of information about that participant's interactions with that peer. However, participants can partially protect themselves by hiding from their peers information those peers don't need, such as whether that participant was the initiator of a particular transaction.

For this purpose, implementations SHOULD ensure that when a participant initiates a transaction, a random number of keys at the start of the path are padding, before the first key that specifies a bilateral agreement with a peer. This SHOULD be accompanied by corresponding random delays before sending messages to those peers; these delays MUST be taken into account when determining deadlines to put in relevant messages. Likewise, the initiator SHOULD simulate such *phantom participants* at the end of the path when creating a complete agreement for the transaction.

If Big Brother's surveillance is accompanied by laws prohibiting mere participation in (as opposed to initiation of) some kinds of circulex transactions (for example, those involving certain currencies), it might become desirable for intermediate participants to include their own phantom participants in paths, too. This won't protect participants who send partial agreements directly to peers who are collaborators, but it might make it harder for Big Brother to make inferences about its collaborators' peers' peers.

A participant's peer acting as a collaborator will be able to reveal to Big Brother the participant's relays, and Big Brother might be able to infer from this that there are likely to be financial trust relationships between the participant and those relays. Therefore, implementations MUST make clear to users that the identities of their chosen relays will be made known to their other peers. Also, implementations that can act as relays MUST make clear

to users that use of this feature will reveal that relationship to the peers of those it acts as relays for.

Via hints and partial and complete agreements, Big Brother will be able to get certain kinds of information generated by participants quite distant from it and any of its collaborators. For example, a characteristic nonstandard currency used in hints might allow Big Brother to infer that multiple transactions are associated with a single participant. Therefore, implementations MUST NOT include highly specific nonstandard currencies in hints without first warning their users and obtaining their consent.

Big Brother might also glean information from the precise deadlines and differences between them in transaction identifiers and bilateral agreements, or even the exchange rates and fees apparent in some paths. Therefore, implementations SHOULD add appropriate amounts of random noise to these numbers, wherever it's possible to do so while maintaining consistency with the protocol and their users' wishes.

As a passive pervasive attacker, Big Brother might be able to observe the timing and sizes of messages sent between peers, even though the end-to-end encryption will prevent it from directly reading the contents of those messages. Participants who want to hide even the fact that their IP addresses are communicating with their peers' IP addresses SHOULD use some form of indirect communication, such as onion routing; implementations SHOULD provide this option for their users.

To make traffic analysis more difficult for Big Brother, implementations MAY combine multiple messages into a single transport-layer transmission. Random delays between receiving a message and sending any messages in response might also make traffic correlation attacks more difficult; such delays, if used, MUST be taken into account when sending latency reports and making other commitments.

6.3 Goliath Corporation

Goliath Corporation wants to extract economic rents from transactions. It might attempt to do so by dominating the circulex network, or, if it already has (or expects to have) dominance in transactions outside circulex, it might try to suppress the adoption of circulex.

6.3.1 Fear, uncertainty, and doubt

If it's taking the latter path, Goliath Corporation is likely to try to portray circulex as unsafe, unusable, and expensive, or as a haven for OSIRIS and other criminals. To prevent such an attack, implementers MUST attempt to

make their implementations as secure and usable as possible, taking into account their intended use cases. Truth is the best possible defence against misinformation about circulex, and Goliath Corporation might try to discredit circulex by criticizing any false statements made in support of it; therefore, promoters of circulex MUST NOT misrepresent or overstate its safety or other features.

6.3.2 Circulex as a service

Goliath Corporation might try to gain dominance within circulex by offering to participate in the circulex network on behalf of users, as a convenience to them, so that they don't have to participate in circulex using their own devices. While there might be legitimate circumstances in which such a service is desirable (to allow participation by people who don't have access to reliable electricity or internet service), it comes at the cost of the users' privacy, and there are further downsides if any one such service provider becomes very popular.

For example, if Goliath Corporation acquired such a service provider, it would be likely to sell access to transaction data, further encroaching on the users' privacy. And even if Goliath Corporation doesn't directly charge its users for the service, its dominance could allow it to impose costs on other circulex participants who want to make payments to (or receive them from) Goliath Corporation's users. When such participants bring this to the attention of the relevant Goliath users, the response might very frequently be something like "Why don't you just get a Goliath account?", thus imposing social pressure in the service of further entrenching Goliath's dominance.

In order to prevent this kind of de facto centralization of the network, circulex-as-a-service software MUST NOT be implemented or deployed in a way or at a time when it's likely to result in any significant number of circulex accounts being controlled by any one person or organization. The self-hosted peer-to-peer model MUST be established as the normal mode of circulex use, wherever this is possible.

6.3.3 Dominance of trust

Even without running circulex instances on behalf of its users, Goliath Corporation could obtain a kind of dominance in the circulex network if it acquires control over circulex instances that have trust relationships with a significant proportion of other circulex instances. This might give Goliath Corporation sufficient transaction data to correlate with other internet traffic data in a way that weakens circulex's privacy properties.

Even if Goliath Corporation controls circulex instances that are trusted only as relays by a significant number of circulex participants, it could try to extract rents from this dominance by refusing to allow its relays to interoperate with other relays that haven't obtained expensive certification. Alternatively, it might try to prevent certain groups of people from using direct trust relationships with people who use Goliath relays.

In order to make the accumulation of such dominance less likely, implementations MUST NOT include any default or suggested relays or instances to establish trust relationships with. However, implementations MAY include suggestions of instances to make donations to or purchase unrelated goods or services from.

6.3.4 Software dominance

Goliath Corporation might also try to exploit the circulex network by dominating the implementations of the protocol.

It's much easier for a single organization to bend software to its own purposes, disregarding the interests of the end users, if that software is proprietary, rather than open source. Therefore, circulex implementations MUST NOT be encumbered by restrictions that inhibit the free use, study, modification, or redistribution of the software.

However, so-called "intellectual property" isn't the only way Goliath might try to dominate circulex software. Goliath might try to distribute its own open source circulex implementation, making it popular by any means necessary, and then develop extensions to the protocol at a pace no other implementers can keep up with, thus achieving an effective monopoly on circulex software. To avoid this outcome, any future version of the circulex protocol, including any optional extensions, MUST remain simple enough that an individual or small team could reasonably be expected to be able to implement it.

6.3.5 Peripheral services

Even if Goliath doesn't directly participate in the circulex network, it might obtain a lot of information from it by offering peripheral services (such as accounting or data storage services) to a large number of organizations. From these relationships, it could extract information about the identities of those organizations' customers and suppliers, and match the data from multiple organizations by inspecting libp2p peer IDs, IP addresses, and sets of relays. Goliath could then exploit this information for its own profit, to the detriment of the privacy of circulex participants.

Such an attack could be mitigated by using a different ephemeral peer ID for each purchase (while maintaining stable peer IDs with long-term peers), and using onion routing for the connection with the vendor, or some other means of hiding the customer's IP address. It might be harder to prevent Goliath from drawing inferences from participants' sets of relays, but for a participant with a large number of potential relays, there might be some advantage in using consistently different sets of relays with different vendors.

6.4 Combined threat

There's another possible scenario worth considering, involving all three threats — OSIRIS, Big Brother, and Goliath Corporation.

Big Brother might point to OSIRIS as a threat, and insist that, in order to prevent OSIRIS from receiving funding and laundering the proceeds of its crimes, all financial transactions must be closely monitored. Big Brother might then establish regulations prohibiting anyone from transacting without the authorization of the Beast (see chapter 13 of [12]).

Goliath is likely to lobby for these regulations to be wide-ranging and difficult to implement, because this will increase the barriers its potential competitors will face in entering the money transfer space. Such regulations would thus make it easier for Goliath to monopolize money transfers, and would allow it to claim that it's not only *permitted* to accumulate vast quantities of data about its customers' behaviour, it's *obliged* to do so.

Such regulations would also partially achieve OSIRIS's denial-of-service goal, by inhibiting the free flow of transactions in places OSIRIS wishes to target.

There's no particularly easy answer to the question of how to defend against such an attack on circulex. Circulex is designed to be resistant to surveillance, as discussed above, but this resistance has its limits. For example, if a particularly authoritarian government bans all use of circulex, and enforces the ban by using random physical spot checks of people's devices, freedom-lovers in that place will need to weigh up the risks and benefits of using circulex.

In places where freedom and democracy are presently valued, a good preemptive defence might involve highlighting circulex's usefulness to people in more authoritarian countries who also value such ideals. In many situations, it might be worth portraying circulex as a piece of infrastructure, like roads, electricity networks, or the internet. While such facilities certainly *can* be used by organizations like OSIRIS for nefarious purposes, cumbersome restrictions on their use would do far more to harm peaceful people than to protect them. It might also be worth ridiculing the idea of heavy regula-

tion and surveillance of friendly IOUs between family members and other acquaintances.

References

[1] Richard Barnes et al. *Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement*. August 2015. RFC 7624.

[2] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. January 2005. STD 66.

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

[4] 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).

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

[6] Scott Bradner and Barry Leiba. *Key words for use in RFCs to Indicate Requirement Levels*. May 2017. BCP 14.

[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] Jasper Fforde. *The Eyre Affair*. Hodder and Stoughton, July 19, 2001.

[9] 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).

[10] Mike Goelzer, Yusef Napora, and Marcin Rataj. *Peer Ids and Keys*. Version r1. August 15, 2019. URL: https://github.com/libp2p/specs/blob/master/peer-ids/peer-ids.md (visited on 2020-09-15).

[11] Jana Iyengar and Martin Thomson, editors. *QUIC: A UDP-Based Multiplexed and Secure Transport*. June 10, 2020. URL: https://datatracker.ietf.org/doc/draft-ietf-quic-transport/ (visited on 2020-09-04).

[12] John. *Revelation*. In: *Holy Bible, New Living Translation*. Translated by Tyndale House Foundation. Tyndale House, 2015. URL: https://www.biblegateway.com/passage/?search=Revelation+13&version=NLT (visited on 2022-09-06).

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

[14] 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).

[15] Barry Leiba. *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. May 2017. RFC 8174.

[16] *libp2p specification*. Protocol Labs. August 31, 2020. URL: https://github.com/libp2p/specs (visited on 2020-09-04).

[17] Andrew Longacre Jr. and Rob Hussey. "Two dimensional data encoding structure and symbology for use with optical readers". US5591956A. January 7, 1997.

[18] Timothy James McKenzie Makarios. *Chunky Base b Encodings*. Version 1.0.0-0.0.1. August 2021. URL: https://circulex.nz/chunky-base-b.html (visited on 2021-08-10).

[19] 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).

[20] 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).

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

[22] *Multiaddr*. The Multiformats Project. January 30, 2020. URL: https://multiformats.io/multiaddr/ (visited on 2020-09-15).

[23] *Multibase*. The Multiformats Project. April 21, 2021. URL: https://github.com/multiformats/multibase (visited on 2021-08-10).

[24] *multicodec/table.csv*. The Multiformats Project. August 21, 2020. URL: https://github.com/multiformats/multicodec/blob/master/table.csv (visited on 2020-09-02).

[25] *Multihash*. The Multiformats Project. May 10, 2020. URL: https://multiformats.io/multihash/ (visited on 2020-09-02).

[26] Yusef Napora. *Connection Establishment in libp2p*. Version r0. June 20, 2019. URL: https://github.com/libp2p/specs/blob/master/connections/README.md (visited on 2020-10-30).

[27] Thomas Narten. *Assigning Experimental and Testing Numbers Considered Useful*. January 2004. BCP 82.

[28] George Orwell. *Nineteen Eighty-Four*. Secker & Warburg, June 8, 1949.

[29] *p2p-circuit relay*. Protocol Labs. June 29, 2021. URL: https://github.com/libp2p/specs/tree/master/relay (visited on 2021-08-18).

[30] Tom Preston-Werner et al. *Semantic Versioning*. Version 2.0.0. June 2020. URL: https://semver.org/spec/v2.0.0.html (visited on 2020-10-30).

[31] Mohammed El-Qorchi. "Hawala". In: *Finance & Development* 39.4 (December 2002). URL: https://www.imf.org/external/pubs/ft/fandd/2002/12/elqorchi.htm (visited on 2022-09-28).

[32] Henry Robinson. *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).

[33] Rumplepay. *About Rumplepay*. URL: https://rumplepay.com/about/ (visited on 2022-09-28).

[34] Markus W. Scherer and Mark Davis. *BOCU-1: MIME-compatible Unicode Compression*. Version 2. February 4, 2006. URL: https://unicode.org/notes/tn6/ (visited on 2021-12-08).

[35] SIX Interbank Clearing AG, editor. *Current currency & funds code list*. URL: https://www.currency-iso.org/en/home/tables/table-a1.html (visited on 2020-06-04).

[36] Stellar Development Foundation. *Stellar - an open network for money*. November 18, 2022. URL: https://stellar.org/ (visited on 2022-11-21).

[37] stellarbeat.io. *Network explorer*. November 21, 2022. URL: https://stellarbeat.io/ (visited on 2022-11-21).

[38] *unsigned-varint*. The Multiformats Project. July 8, 2020. URL: https://github.com/multiformats/unsigned-varint (visited on 2020-09-02).

[39] Henry de Valence, Isis Lovecruft, and Tony Arcieri. *The Ristretto Group*. URL: https://ristretto.group/ (visited on 2020-03-17).

[40] Ian Vásquez and Tanja Porcnik. *Human Freedom Index*. 2019. URL: https://www.cato.org/human-freedom-index-new (visited on 2020-09-11).

[41] *X.680 : Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation*. The International Telecommunications Union — Telecommunication Standardization Sector. August 13, 2015. URL: https://handle.itu.int/11.1002/1000/12479 (visited on 2020-02-14).

[42] *X.681 : Information technology — Abstract Syntax Notation One (ASN.1): Information object specification*. The International Telecommunications Union — Telecommunication Standardization Sector. August 13, 2015. URL: https://handle.itu.int/11.1002/1000/12480 (visited on 2020-02-21).

[43] *X.682 : Information technology — Abstract Syntax Notation One (ASN.1): Constraint specification*. The International Telecommunications Union — Telecommunication Standardization Sector. August 13, 2015. URL: https://handle.itu.int/11.1002/1000/12481 (visited on 2020-02-14).

[44] *X.696 : Information technology — ASN.1 encoding rules: Specification of Octet Encoding Rules (OER)*. The International Telecommunications Union — Telecommunication Standardization Sector. August 13, 2015. URL: https://handle.itu.int/11.1002/1000/12487 (visited on 2020-02-26).

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