A Protocol for Interledger Payments

Stefan Thomas & Evan Schwartz
{stefan,evan}@ripple.com

Abstract

We present a protocol for payments across payment systems. It enables secure transfers between *ledgers* and allows anyone with accounts on two ledgers to create a connection between them. Ledger-provided escrow removes the need to trust these *connectors*. Connections can be composed to enable payments between any ledgers, creating a global graph of liquidity or *Interledger*.

Unlike previous approaches, this protocol requires no global coordinating system or blockchain. Transfers are escrowed in series from the sender to the recipient and executed using one of two modes. In the *Atomic* mode, transfers are coordinated using an ad-hoc group of *notaries* selected by the participants. In the *Universal* mode, there is no external coordination. Instead, bounded execution windows, participant incentives and a "reverse" execution order enable secure payments between parties without shared trust in any system or institution.

1 Introduction

Sending money today within any single payment system is relatively simple, fast and inexpensive. However, moving money between systems is cumbersome, slow and costly, if it is possible at all.

Digital payment systems use *ledgers* to track accounts and balances and to enable local transfers between their users. Today, there are few *connectors* facilitating payments between these ledgers and there are high barriers to entry for creating new connections. Connectors are not standardized and they must be trusted not to steal the sender's money.

In this paper, we present a protocol for *interledger* payments that enables anyone with accounts on two ledgers to create connections between them. It uses ledger-provided *escrow*—conditional locking of funds—to allow secure payments through untrusted connectors.

Any ledger can integrate this protocol simply by enabling escrowed transfers. Unlike previous approaches, our protocol does not rely on any global coordinating system or ledger for processing payments—centralized [10] or decentralized [15, 17, 19].

Our protocol lowers the barriers to facilitating interledger payments. Connectors compete to provide the best rates and speed. The protocol can scale to handle unlimited payment volume, simply by adding more connectors and ledgers. Composing connectors into chains enables payments between any ledgers and give small or new payment systems the same network effects as the most established systems. This can make every item of value spendable anywhere—from currencies to commodities, computing resources and social credit.

Our protocol provides:

- Secure payments through any connector using ledger-provided escrow.
- The sender of a payment is guaranteed a cryptographically signed receipt from the recipient, if the payment succeeds, or else the return of their escrowed funds.
- Two modes of executing payments: In the *Atomic* mode, transfers are coordinated by an ad-hoc group of *notaries* selected by the participants to ensure all transfers either execute or abort. The *Universal* mode instead uses bounded execution windows and incentives to remove the need for any mutually trusted system or institution.

The following sections describe our protocol in greater depth. Section 2 introduces the core concepts of the protocol, including the actors involved and the need for ledger-provided escrow. Section 3 presents the *Atomic* mode. Section 4 presents the *Universal* mode. Sequence diagrams and the formal protocol specifications can be found in the Appendix.

2 Interledger Payments

A *ledger* may be used to track anything of value—from currency or stocks to physical goods and titles—and may be centralized or decentralized [17]. As illustrated in Figure 1, payments between accounts in the same system are accomplished with a simple *book transfer*.

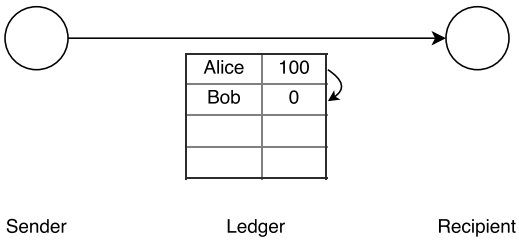

Figure 1: Book transfer

A *connector* is a system that facilitates interledger payments by coordinating book transfers on multiple ledgers. Connectors can also translate between different protocols used by these ledgers. As illustrated in Figure 2, connector *Chloe* accepts a transfer into her account on one ledger in exchange for a transfer out of her account on another ledger.

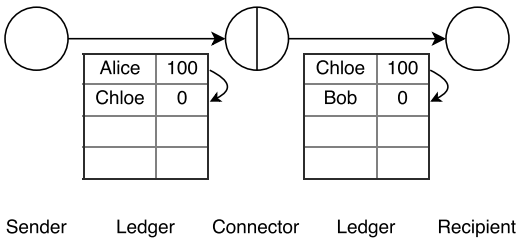

Figure 2: Connector controls accounts on two ledgers

The problem with existing payment protocols is that the sender must trust the connector to follow through on paying the intended recipient. Nothing technically prevents connectors from losing or stealing money, so they must be bound by reputation and legal contracts to complete the payment correctly. This severely limits the set of institutions that can act as connectors, resulting in a highly uncompetitive and disconnected global payment system.

Ledger-provided escrow guarantees the sender that their funds will only be transferred to the connector once the ledger receives proof that the recipient has been paid. Escrow also assures the connector that they will receive the sender's funds once they complete their end of the agreement.

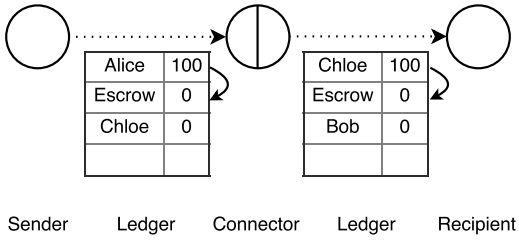

Figure 3: Funds are escrowed by the sender and connector on their respective ledgers

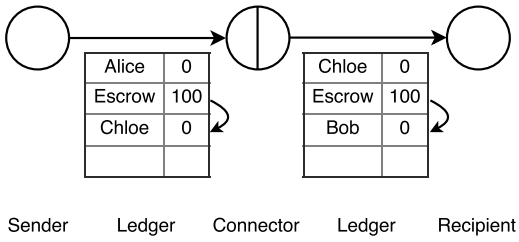

Figure 4: The transfers to the connector and recipient are either executed or aborted together

The simple arrangement illustrated here can be extended into arbitrarily long chains of connectors to facilitate payments between any sender and any recipient. This brings the small-world network effect (commonly known as the "six degrees of separation") [3, 21] to payments and creates a global graph of liquidity or *Interledger*.

2.1 Byzantine and Rational Actors

A truly open protocol for payments cannot rely on highly trusted actors for security. It must lower the barriers to participation through built-in protections against potentially faulty, self-interested, or malicious behavior by the participants.

We use the Byzantine, Altruistic, Rational (BAR) model introduced by [2] for categorizing participants. *Byzan-*

tine actors may deviate from the protocol for any reason, ranging from technical failure to deliberate attempts to harm other parties or simply impede the protocol. *Altruistic* actors follow the protocol exactly. *Rational* actors are self-interested and will follow or deviate from the protocol to maximize their short and long-term benefits.

We assume all actors in the payment are either Rational or Byzantine. Protocols that rely on highly trusted actors effectively assume all to be Altruistic—or bound by factors external to the protocol. Assuming the actors to be self-interested or malicious requires the protocol to provide security even when participants are bound only by the algorithm itself.

We assume that, as Rational actors, ledgers and connectors may require some fee to incentivize their participation in a payment. These fees may be paid in the flow of the payment, or out of band. In this paper, we will only discuss the details of fees necessary to mitigate risks and attacks specific to interledger payments.

While ledgers may be Byzantine, we do not attempt to introduce fault-tolerance—protection against accidental or malicious errors—for participants who hold accounts with such a ledger. Ledgers themselves can be made Byzantine fault-tolerant [15, 17, 19], and participants can choose the ledgers with which they hold accounts. We only seek to isolate participants who hold accounts on non-faulty ledgers from risk.

We assume that connectors will agree to participate in a payment only if they face negligible or manageable risk for doing so, and that they will charge fees accordingly. Connectors will only deliver money on a destination ledger if doing so benefits them directly.

Any of the participants in a payment may attempt to overload or defraud any of the other actors involved. Thus, escrow is needed to make secure interledger payments.

2.2 Cryptographic Escrow

Ledger-provided escrow enables secure interledger payments by isolating each participant from the risks of failure or malicious behavior by others involved in the payment. It represents the financial equivalent of the states in the Two-Phase Commit Protocol [13, 14].

Providing escrowed transfers is the main requirement for ledgers to enable secure interledger payments.

All participants rely upon their ledgers to escrow funds and release them only when a predefined condition is met. The sender is assured by their ledger that their funds are transferred only upon delivery of a non-repudiable acknowledgement that the recipient has received their payment. The recipient is assured by their ledger that they will be paid when they provide such an acknowledgement. Connectors also use their ledgers' escrow to protect themselves from risk.

Cryptographic signatures are a simple way for ledgers to securely validate the outcome of the external conditions upon which a transfer is escrowed. Any one-way function can be used [18]. Using asymmetric cryptography, the ledger escrows funds pending the presentation of a valid signature for a pre-defined public key and message or hash. The ledger can then easily validate the signature when it is presented and determine if the condition has been met.

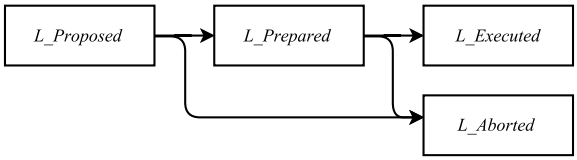

Figure 5: Transfer states

Figure 5 illustrates the states of a transfer. First, a transfer is *proposed* to the participants, but no changes are made to the ledger. Once affected account holders have authorized the transfer, the ledger checks that funds are available and that all of its rules and policies have been satisfied. The ledger places the funds in escrow and the transfer is *prepared*. If the execution condition e is fulfilled, the transfer is *executed*. If any of the checks fail, or if an abort condition e' is fulfilled, the transfer is *aborted* and the escrowed funds are returned to their originator.

3 Atomic Interledger Payments

Interledger payments consist of transfers on different ledgers. If the payment partially executes, at least one of the participants loses money. Thus, participants want *atomicity*—a guarantee that all of the component transfers will either execute or that all will be aborted.

Executing transfers atomically across multiple ledgers requires a *transaction commit protocol*. The simplest such protocol is the Two-Phase Commit, in which all systems involved first indicate their readiness to complete the transaction, and then execute or abort based on whether all of the other systems have agreed. The basic form of that protocol uses a *transaction manager* to collect the responses of the various system and disseminate the decision. Fault tolerance can be added to the Two-Phase Commit by replacing the single transaction

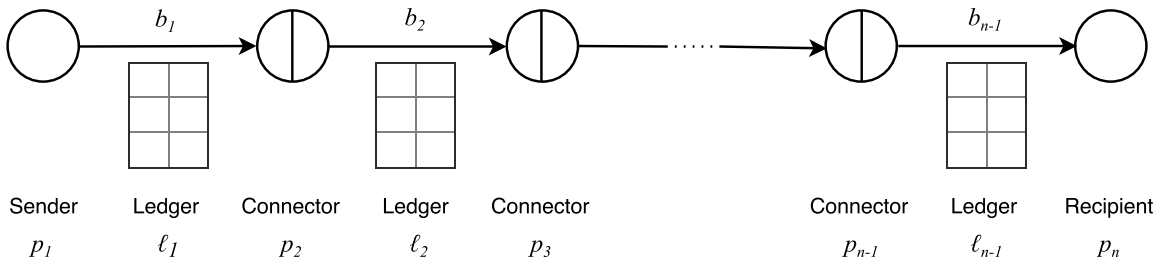

Figure 6: Payment chain

manager with a group of coordinators that use a consensus algorithm to agree on the outcome of the payment [14].

The Atomic execution mode uses a Byzantine fault-tolerant agreement algorithm amongst an ad-hoc group of coordinators or *notaries* to synchronize the execution of the component transfers. Additionally, it guarantees that the sender receive a cryptographically signed payment receipt from the recipient if funds are transferred.

Figure 6 shows a payment through a chain of participants P and ledgers L. There are n participants in P, such that $|P| = n$. The sender is the first participant p_1 and the recipient is the last participant p_n. The connectors C are participants p_2 through p_{n-1}, such that $\{p_i \in C \mid C \subset P \wedge i \in \mathbb{Z}^+ \wedge 1 < i < n\}$. The payment consists of $n-1$ book transfers B on $n-1$ ledgers, such that $|B| = |L| = n-1$. The first participant, the sender p_1, has an account on ledger ℓ_1. The last participant, the recipient p_n, has an account on ledger ℓ_{n-1}. Each connector $p_i \in C$ has accounts on ledgers ℓ_{i-1} and ℓ_i and facilitates payments between them.

3.1 Notaries

In the Atomic mode, transfers are coordinated by a group of *notaries* N that serve as the source of truth regarding the success or failure of the payment. They take the place of the transaction manager in a basic Two-Phase Commit. Importantly, notaries are organized in ad-hoc groups for each payment and our protocol does not require one globally trusted set of notaries.

All of the escrow conditions for the book transfers B comprising the payment must depend on the notaries either sending a *D_Execute* or *D_Abort* message. If the escrow conditions were based solely on a message from N, however, faulty notaries could cause the transfers to execute before the final transfer to the recipient is *L_Prepared*. [11].

Notaries N must only agree on the *D_Execute* or *D_Abort* messages if they have received a signed receipt, the *R_ReceiptSignature*, from the recipient p_n before a timeout t. The recipient's signature provides non-repudiable proof that they have been paid. The recipient p_n signs the receipt once the transfer into their account is *L_Prepared* and escrowed pending their signature.

To ensure all of the transfers B can be executed atomically, all of the execution conditions E must depend on the *D_Execute* message from the notaries and the *R_ReceiptSignature* from the recipient:

$$\forall e \in E : e = R\_ReceiptSignature \wedge D\_Execute \quad (1)$$

The abort conditions E' for each transfer in B are dependent only on the abort message *D_Abort*. Receipt of a message fulfilling the abort condition e'_i where $\{i \in \mathbb{Z}^+ \wedge i < n\}$ causes the ledger ℓ_i to immediately transition the *L_Proposed* or *L_Prepared* transfer b_i to the *L_Aborted* state and release the funds to the originator.

$$\forall e' \in E' : e' = D\_Abort \quad (2)$$

3.2 Fault Tolerance

The Atomic mode only guarantees atomicity when notaries N act honestly. Like all other participants, we assume notaries are either rational or Byzantine. Rational actors can be incentivized to participate with a fee.

Byzantine notaries, however, could sign both *D_Execute* and *D_Abort* messages, communicate them to different ledgers and cause some transfers to be executed and other to be aborted. In order to protect against this, we must set a fault-tolerance threshold f. This means that the outcome of the agreement protocol will be correct so long as there are no more than f Byzantine notaries in N.

If $f = 0$, we only need a single notary $N = \{N_1\}$ and the *D_Execute* and *D_Abort* messages are simply signatures by that notary N_1.

If $f \geq 1$, notaries use a Byzantine fault-tolerant (BFT) agreement protocol. Using the method from [14, 16], a BFT replication algorithm, such as PBFT [8] or Tangaroa, a BFT version of Raft [9], can be simplified into a binary agreement protocol.

The minimum number of processes required to tolerate f Byzantine faults is $|N| = 3f + 1$ as shown by [5].

The *D_Execute* and *D_Abort* messages are collections of signatures from some representative subset N_{rep} such that $N_{rep} \subseteq N$ and $|N_{rep}| = f + 1$ vouching for the outcome of the agreement protocol.

3.3 Timeout

To ensure that funds cannot be held in escrow forever, even in the case of failure, notaries N enforce a timeout t. The recipient p_n must submit the *R_ReceiptSignature* to the notaries N before t is reached, or the payment will be aborted and the escrowed funds returned. t must be sufficient to account for the duration of the phases of the protocol leading up to Execution.

3.4 Phases of the Protocol

Before a payment can occur, the sender p_1 must find a suitable set of connectors C forming a path to the recipient p_n. Connectors have an interest in making their liquidity information available in order to attract payment flows. The problems of minimum-cost and multicommodity flow have been studied extensively in the context of planning [1, 7, 20].

In the following, we assume that a path has already been chosen and the exchange rates and any fees quoted by the connectors in C are known.

Appendix A.1 illustrates the phases of the protocol, excluding Notary Selection and Notary Setup.

3.4.1 Notary Selection

Notaries are selected by the participants P.

For each candidate fault tolerance threshold f_c where $f_c \in Z^+ \wedge f_c < f_{max}$ and f_{max} is the sender's maximum fault tolerance threshold, the sender requests the set of all notaries trusted at the given fault tolerance threshold f_c from each participant p, $N_p(f_c)$ and calculates the candidate set $N_c(f_c)$ at threshold f_c as the intersection of these sets. Each $p \in P$ chooses $N_p(f_c)$ such that they believe that there is no Byzantine subset N_{evil} where $N_{evil} \subseteq N_p(f_c), |N_{evil}| > f$ which will collude against them.

$$\forall f_c \in Z^+ \wedge f_c < f_{max} : N_c(f_c) = \bigcap_{p \in P} N_p(f_c) \quad (3)$$

Finally the sender chooses a fault tolerance threshold f and a corresponding set of notaries N such that $N \subseteq N_c(f) \wedge |N| \geq 3f + 1$. If no such set exists, the sender cannot rely on the Atomic mode and must instead use the Universal mode as described in Section 4.

3.4.2 Proposal

In the proposal phase, the sender p_1 notifies each connector $\{p_i \mid i \in \mathbb{Z}^+ \wedge 1 < i < n\}$ about the the book transfers b_{i-1} and b_i in the upcoming payment. Upon receiving the proposal, each connector p_i will verify the proposed spread between the payments matches its exchange rate, charge its fee and store the payment details. p_i accepts the terms of the book transfers b_{i-1} and b_i and the sender p_1 proceeds to the next phase.

3.4.3 Preparation

Unlike in a basic Two-Phase Commit, book transfers $\{b_i \in B \mid i \in \mathbb{Z}^+ \wedge 1 < i < n\}$ are prepared in sequence from b_1 to b_{n-1}. Each connector is only willing to escrow their funds if they know funds have already been escrowed for them.

The sender p_1 first authorizes and sends the instruction to prepare the first transfer b_1 on ℓ_1. p_1 then requests that the first connector p_2 prepare b_2 on ℓ_2. The connector p_2 is comfortable preparing b_2 because b_1 is prepared and the funds have been escrowed by ℓ_1. Similarly, each connector p_i prepares transfer b_i once it is notified that ℓ_{i-1} has prepared b_{i-1} and escrowed the corresponding funds.

3.4.4 Execution

Once the last book transfer b_{n-1} has been prepared and the funds escrowed, the recipient p_n must sign the receipt before the timeout t. p_n is comfortable signing the receipt because they know that doing so will fulfill the condition of the escrowed funds waiting for them on ℓ_{n-1}. p_n submits the *R_ReceiptSignature* to the notaries

N, who then run the agreement protocol (see Section 3.2) to decide whether the payment should execute.

If N agree that the *R_ReceiptSignature* was received in time, they submit it and a signed *D_Execute* message to all participants P. Each participant $\{p_i \in P \mid i \in \mathbb{Z}^+ \wedge 1 < i \leq n\}$ submits the *R_ReceiptSignature* and *D_Execute* message to ℓ_{i-1} to execute b_{i-1} and claim the funds they are due.

If N agree that the payment timed out, they submit the a signed *D_Abort* message to P. Each participant $\{p_j \in P \mid j \in \mathbb{Z}^+ \wedge 1 \geq j < n\}$ submits the *D_Abort* message to ℓ_j and reclaims its escrowed funds.

3.5 Correctness

The Atomic mode of the protocol inherits the assumptions and level of fault-tolerance from the Byzantine agreement protocol used amongst the notaries. For the purpose of this section, we assume the notaries use PBFT [8].

Given the assumptions in [8] we require no additional assumptions.

3.5.1 Safety

Safety means that if one non-faulty ledger transitions its transfer to the *L_Executed* state, then no non-faulty ledger will transition a transfer belonging to the same payment to the *L_Aborted* state and vice versa. If liveness also holds, all transfers will eventually be executed or all aborted.

Given the safety of the consensus algorithm used by the notaries, we know that all ledgers will only receive one of either: $f+1$ *D_Execute*, or $f+1$ *D_Abort* messages from notaries. A correct ledger only executes the transfer when it has received $f+1$ *D_Execute* messages which precludes the possibility that any other correct ledger has aborted their transfer. Equally, a correct ledger only aborts the transfer when it has received $f+1$ *D_Abort* messages which precludes the possibility that any other correct ledger has received $f+1$ *D_Execute* messages and aborted their transfer.

3.5.2 Liveness

Liveness means that every non-faulty ledger connected to at least one non-faulty, rational participant will eventually execute or abort its transfer.

As mentioned in Section 3.3, the notaries will initiate their agreement protocol spontaneously after a timeout t. From the liveness property of the underlying agreement protocol, we know that the notaries will eventually decide to either *D_Execute* or *D_Abort* and broadcast that decision to the participants. Each ledger is connected to at least two participants. If one of these participants is non-faulty, it will forward the decision to the ledger.

Note that if notaries could broadcast directly to the ledgers, our protocol could maintain liveness even when all participants are faulty. However, in real world applications, some ledgers use proprietary protocols or private networks, so we cannot rely on the fact that the notaries can reach them directly.

If a transfer is in the *L_Prepared* state, at least one of the participants has an interest in seeing the transfer reach a final state, because the escrowed funds would be transferred to them. If this participant is rational they will therefore eventually forward the notaries' decision. A Byzantine participant in this position may not, but in doing so does not hurt anyone else.

3.6 Fees

Each connector in C incurs some costs and risks for participating in a payment, which can be priced into the connectors' fees.

When trading different assets, connectors in C effectively write the sender p_1 an *American option* [4, 6] which, on exercise, swaps an asset on one ledger for an equivalent asset on another ledger. In addition to the factors considered in standard option pricing, the connector should also account for the following attack in its fee.

3.6.1 Liquidity Starvation

An attacker can attempt to temporarily tie up all of a connector's liquidity in payments it knows will fail. This attack is rendered uneconomical if the connector sets its fee to cover its costs and the profit it would expect if the payment were successful. Furthermore, connectors, including those operated by the same entity as a ledger, can prevent their funds from being completely tied up by escalating their fees as a function of the percentage of their total liquidity being held in escrow.

4 Universal Interledger Payments

While the Atomic mode uses notaries to ensure proper execution of a payment, the Universal mode relies on the incentives of rational participants instead to eliminate the need for external coordination. It provides safety and liveness for all non-faulty participants connected to only non-faulty ledgers, under an assumption of bounded synchrony with a known bound. In Section 4.4.1 we discuss the practical considerations of using Universal mode in a system that does not guarantee bounded synchrony, e.g. the Internet.

Appendix A.2 illustrates the phases of the payment in this mode.

4.1 Execution Order

In Universal mode, there are no notaries. The book transfers B must be executed in a specific order to ensure all participants' incentives are aligned to execute the payment properly and to ensure delivery of the *R_ReceiptSignature* to the sender p_1.

The transfers B are escrowed only on the condition of receiving the *R_ReceiptSignature*:

$$\forall e \in E : e = R\_ReceiptSignature \tag{4}$$

Instead of having a global timeout, each book transfer in $\{b_i \in B \mid i \in \mathbb{Z}^+ \wedge i < n\}$ has its own expiration time enforced by the ledger ℓ_i. After the last book transfer b_{n-1} is prepared, the recipient p_n signs the receipt and presents the *R_ReceiptSignature* directly to their ledger ℓ_{n-1}. If it is before the transfer's expiration time t_{n-1}, b_{n-1} will be executed immediately.

Once b_{n-1} is executed and the recipient p_n is paid, connector p_{n-1} has a very strong incentive to pass the *R_ReceiptSignature* back to ℓ_{n-2}, as they have paid out money but have not yet been paid. When each connector $\{p_j \in P \mid j \in \mathbb{Z}^+ \wedge 1 < j < n\}$ learns of the execution of the book transfer b_j, they must get the *R_ReceiptSignature* from ℓ_j and submit it to ℓ_{j-1} to claim the money waiting in escrow for them.

Thus, the transfers in B are executed in "backwards" order, from the recipient p_n to the sender p_1. Once the first transfer b_1 is executed, the sender p_1 can get the *R_ReceiptSignature* from their ledger ℓ_1.

If the last transfer b_{n-1} times out before p_n submits the *R_ReceiptSignature*, all transfers in B will expire and the escrowed funds will be returned to their originator. The following section discusses expiration times and the message delay risk connectors must manage.

4.2 Message Delay

Each book transfer in B must have an expiration time t to ensure liveness. In order for a connector $\{p_i \in P \mid i \in \mathbb{Z}^+ \wedge 1 < i < n\}$ to agree to take part in the payment, they must be able to pass the *R_ReceiptSignature* from ledger ℓ_i to ℓ_{i-1} and execute b_{i-1} before it expires. If b_i is executed but b_{i-1} expires, p_i loses money. Because b_i may execute very close to its expiration time t_i, the expiration time t_{i-1} for transfer b_{i-1} must be greater than that of b_i by some finite time difference $t_{i-1} - t_i$.

This time difference $t_{i-1} - t_i$ must account for the messaging delays $M(\ell_i, p_i)$ from ℓ_i to p_i and $M(p_i, \ell_{i-1})$ from p_i to ℓ_{i-1} (which includes the processing delays at p_i, ℓ_i and ℓ_{i-1} respectively) and the clock skew $K(\ell_{i-1}, \ell_i)$ between ledgers ℓ_{i-1} and ℓ_i.

$$t_i \geq t_{i+1} + M(\ell_i, p_{i+1}) + M(p_{i+1}, \ell_i) + S(\ell_i, \ell_{i+1}) \tag{5}$$

The timeout t_{n-1} for the destination transfer b_{n-1} is equal to the timeout t in Atomic mode introduced in Section 3.3. That is, it is large enough to allow for the preparation of the transfers and for the recipient to sign and submit the *R_ReceiptSignature*.

4.3 Correctness

For our analysis of Universal mode, we consider both safety and liveness under bounded synchrony. We define bounded synchrony similar to the definition given in [12] as a system in which there is a known upper bound M on messaging delays between processes and a known upper bound S on clock skew between two nodes. We assume that processing times are negligible and included in M.

4.3.1 Safety

Safety means that there will be no book transfer $\{b_i \in B \mid i \in \mathbb{Z}^+ \wedge i < n-1\}$ which expires if b_{i+1} executed unless ℓ_i, ℓ_{i+1} or participant p_{i+1} are faulty.

Let b_{i+1} execute at time ϕ_0. We know that b_{i+1} was not expired, therefore:

$$\phi_0 < t_{i+1} \tag{6}$$

A correct ledger ℓ_{i+1} will send a message $\langle M\_ExecuteNotify, R\_ReceiptSignature\rangle$ to connector p_{i+1} which will arrive at time ϕ_1:

$$\phi_1 \leq \phi_0 + M(\ell_i, p_{i+1}) \tag{7}$$

The rational connector p_{i+1} will send a message $\langle M\_ExecuteRequest, R\_ReceiptSignature\rangle$ to ledger ℓ_i which will arrive at time ϕ_2.

$$\phi_2 \leq \phi_1 + M(p_{i+1}, \ell_i) \tag{8}$$

The correct ledger ℓ_i is guaranteed to execute transfer b_i if and only if the message arrives before the timeout t_i. However, so far we have expressed all times in terms of ℓ_{i+1}'s local clock. In order to ensure that b_i is not expired, we must account for clock skew:

$$t_i \geq \phi_2 + S(\ell_i, \ell_{i+1}) \tag{9}$$

From equations (5), (6), (7), (8) and (9):

$$\begin{aligned} t_i &\geq \phi_2 + S(\ell_i, \ell_{i+1}) \\ &\geq \phi_1 + M(p_{i+1}, \ell_i) + S(\ell_i, \ell_{i+1}) \\ &\geq \phi_0 + M(\ell_i, p_{i+1}) + M(p_{i+1}, \ell_i) + S(\ell_i, \ell_{i+1}) \\ &\geq t_{i+1} + M(\ell_i, p_{i+1}) + M(p_{i+1}, \ell_i) + S(\ell_i, \ell_{i+1}) \end{aligned} \tag{10}$$

Since ℓ_i is a correct ledger, it will execute the transfer. A transfer that has been executed on a correct ledger cannot expire, therefore b_i cannot expire.

4.3.2 Liveness

Liveness means that eventually each book transfer $\{b_i \in B \mid i \in \mathbb{Z}^+ \wedge i < n\}$ on a non-faulty ledger ℓ_i must either be *L_Executed* or *L_Aborted*.

All book transfers b_i have a finite expiry time t_i. A correct ledger ℓ_i will expire transfer b_i at time t_i unless it has already executed.

Therefore after time t_i, transfer b_i will be either *L_Executed* or *L_Aborted*.

4.4 Fault and Attack Mitigation

In Section 3.6 we discussed the connector fee in Atomic mode. In Universal mode, there are additional costs that the connector must take into account:

4.4.1 Optimal Timeouts

In an asynchronous system, message delays and clock skew are unbounded, so a connector $\{p_i \in P \mid i \in \mathbb{Z}^+ \wedge 1 < i < n\}$ may lose the race to forward the *R_ReceiptSignature* from ℓ_i to ℓ_{i-1}, causing them to lose money.

However, by observing prior performance of the network, p_i can estimate the probability $\Pr(t_{i-1} - t_i \geq M(\ell_i, p_{i+1}) + M(p_{i+1}, \ell_i) + S(\ell_i, \ell_{i+1}))$, calculate the value of the risk to them and include it in their fee.

As $t_{i-1} - t_i$ becomes larger, the risk decreases. However, the expiration time for each transfer $\{b_j \in B \mid j \in \mathbb{Z}^+ \wedge j < i-1\}$ also increases. Longer expiration times incur higher fees, because funds may be held in escrow for a longer period. The sender p_1 will try to choose $t_{i-1} - t_i$ such that the total amount of fees is minimized.

4.4.2 Robust Messaging

The other participants may collude in an attempt to defraud a connector. In order to do so, they must interfere with the messaging as mentioned in the previous section to prevent the connector $\{p_i \in P \mid i \in \mathbb{Z}^+ \wedge 1 < i < n\}$ from completing the transfer b_{i-1} thereby profiting at p_i's expense.

The mitigation for this attack varies based on the technical characteristics of each ledger. However, in all cases, it is a type of Denial of Service mitigation. Both connectors and ledgers have an interest in establishing reliable communication.

4.4.3 Receipt Privacy

The recipient p_n does not have to transfer any money, but we assume that the act of signing the receipt has some external meaning, such as nullifying an existing asset of the recipient (e.g. an invoice) or creating a new liability.

To claim the funds escrowed for them, the recipient p_n must submit *R_ReceiptSignature* to ℓ_{n-1} before the transfer b_{n-1} is executed. In an asynchronous system, b_{n-1} might timeout while this message is in transit.

In order to guarantee safety for the recipient then, we must introduce another property of ledgers, *ReceiptPrivacy*. A correct ledger that offers *ReceiptPrivacy* does not disclose a $\langle D\_Execute, R\_ReceiptSignature\rangle$ message (or the *R_ReceiptSignature* contained therein) unless b_i is executed successfully.

If the recipient chooses an honest ledger with *ReceiptPrivacy*, they are not at risk, even in an asynchronous system, because their ledger ℓ_{n-1} will disclose *R_ReceiptSignature* if and only if b_{n-1} executes.

5 Conclusion

We have proposed a protocol for secure interledger payments across an arbitrary chain of ledgers and connectors. It uses ledger-provided escrow based on cryptographic conditions to remove the need to trust the connectors.

The Atomic mode of the protocol provides atomicity for payment chains in which the participants can agree upon a group of notaries. The Universal mode uses the incentives of rational actors to enable practical payments between participants that do not all share trust in any institution or system.

Our protocol does not rely on any single system for processing payments, so there is no limit to its scalability. Payments can be as fast and cheap as the constituent ledgers and connectors allow and transaction details are private to their participants. The separation of concerns and the minimal standardization requirements enable continuous optimization and competition between connectors and between ledgers.

Removing the need to trust the connector enables anyone with accounts on two or more ledgers to make connections between them. Connectors can be composed to make payments and the financial system more accessible, competitive and resilient. This enables the creation of a global graph of liquidity or *Interledger*.

References

[1] AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. B. Network flows. Tech. rep., DTIC Document, 1988.

[2] AIYER, A. S., ALVISI, L., CLEMENT, A., DAHLIN, M., MARTIN, J.-P., AND PORTH, C. Bar fault tolerance for cooperative services. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 45–58.

[3] ALBERT, R., JEONG, H., AND BARABÁSI, A.-L. Internet: Diameter of the world-wide web. *Nature 401*, 6749 (1999), 130–131.

[4] BLACK, F., AND SCHOLES, M. The pricing of options and corporate liabilities. *The journal of political economy* (1973), 637–654.

[5] BRACHA, G., AND TOUEG, S. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM) 32*, 4 (1985), 824–840.

[6] BRENNAN, M. J., AND SCHWARTZ, E. S. The valuation of american put options. *Journal of Finance* (1977), 449–462.

[7] CAI, X., SHA, D., AND WONG, C. Time-varying minimum cost flow problems. *European Journal of Operational Research 131*, 2 (2001), 352–374.

[8] CASTRO, M., LISKOV, B., ET AL. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[9] COPELAND, C., AND ZHONG, H. Tangaroa: a byzantine fault tolerant raft.

[10] DAVIES, D. W., AND PRICE, W. L. *Security for computer networks: and introduction to data security in teleprocessing and electronic funds transfer*. John Wiley & Sons, Inc., 1989.

[11] DOLEV, D., AND STRONG, H. R. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing 12*, 4 (1983), 656–666.

[12] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*, 2 (1988), 288–323.

[13] GRAY, J. Notes on data base operating systems. In *Operating Systems, An Advanced Course* (London, UK, UK, 1978), Springer-Verlag, pp. 393–481.

[14] GRAY, J., AND LAMPORT, L. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS) 31*, 1 (2006), 133–160.

[15] MAZIÈRES, D. The stellar consensus protocol: A federated model for internet-level consensus.

[16] MOHAN, C., STRONG, R., AND FINKELSTEIN, S. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In *Proceedings of the second annual ACM symposium on Principles of distributed computing* (1983), ACM, pp. 89–103.

[17] NAKAMOTO, S. Bitcoin: A Peer-to-Peer electronic cash system.

[18] ROMPEL, J. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing* (1990), ACM, pp. 387–394.

[19] SCHWARTZ, D., YOUNGS, N., AND BRITTO, A. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper* (2014).

[20] WAGNER, H. M. On a class of capacitated transportation problems. *Management Science 5*, 3 (1959), 304–318.

[21] WATTS, D. J., AND STROGATZ, S. H. Collective dynamics of small-worldnetworks. *nature 393*, 6684 (1998), 440–442.

A Appendix

A.1 Atomic Mode Sequence Diagram

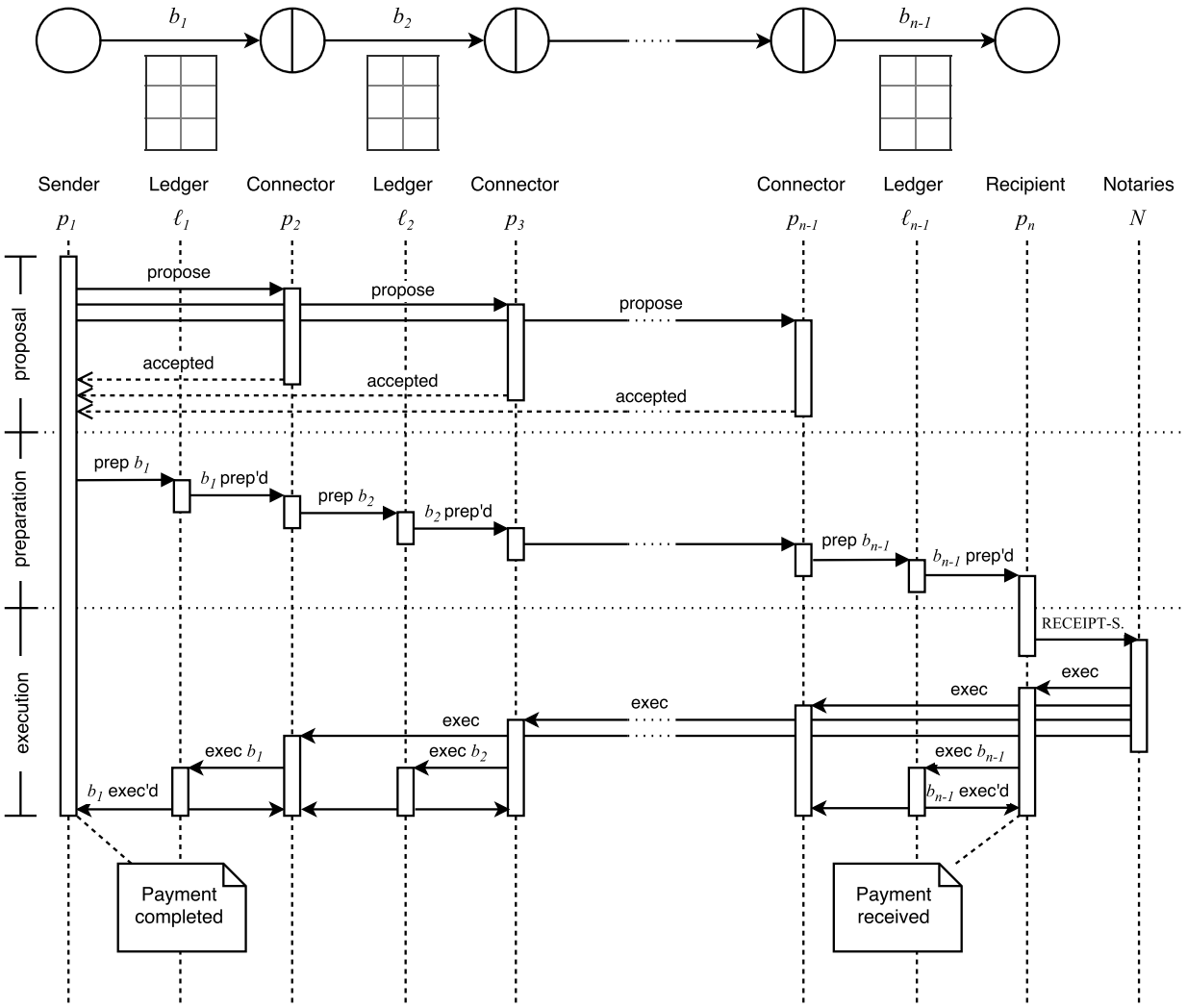

Figure 7: Phases of a payment in the Atomic mode

A.2 Universal Mode Sequence Diagram

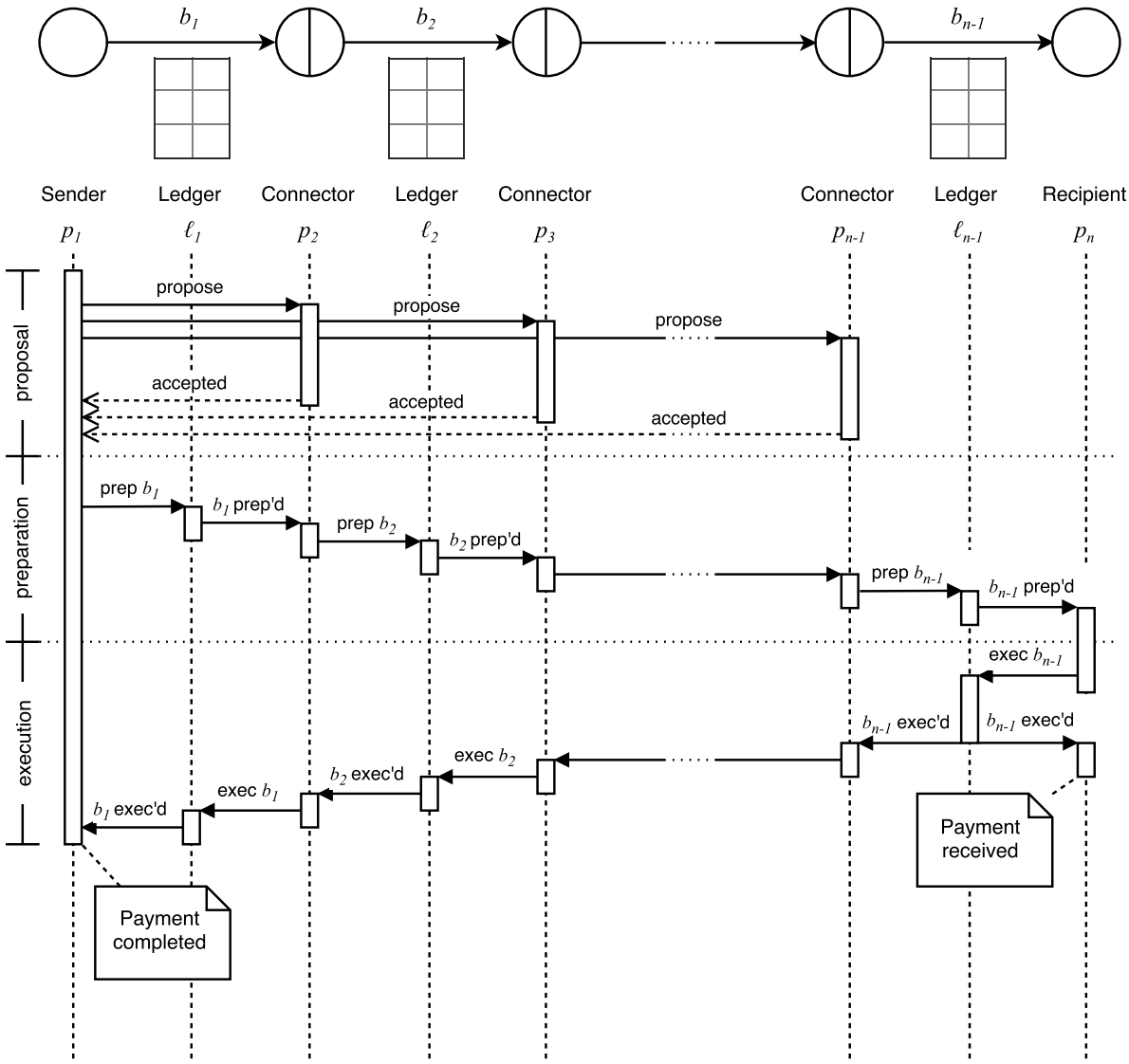

Figure 8: Phases of a payment in the Universal mode

A.3 Formal Specification: Atomic mode

MODULE *Atomic*

Formal Specification in TLA$^+$ of the Interledger Protocol *Atomic* (*ILP*/A)

```
Modeled after the excellent Raft specification by Diego Ongaro.
  Available at https://github.com/ongardie/raft.tla

  Copyright 2014 Diego Ongaro.
  This work is licensed under the Creative Commons Attribution-4.0
  International License https://creativecommons.org/licenses/by/4.0/
```

EXTENDS *Naturals*, *Sequences*, *Bags*, *TLC*

The set of ledger *IDs*
CONSTANTS *Ledger*

The set of participant *IDs*
CONSTANTS *Participant*

The notary
CONSTANTS *Notary*

Sender states
CONSTANTS *S_Ready*, *S_Waiting*, *S_Done*

Notary states
CONSTANTS *N_Waiting*, *N_Committed*, *N_Aborted*

Ledger states
CONSTANTS *L_Proposed*, *L_Prepared*, *L_Executed*, *L_Aborted*

Message types
CONSTANTS *PrepareRequest*, *ExecuteRequest*, *AbortRequest*,
PrepareNotify, *ExecuteNotify*, *AbortNotify*,
SubmitReceiptRequest

Receipt signature
CONSTANTS *R_ReceiptSignature*

Global variables

A bag of records representing requests and responses sent from one process
to another
VARIABLE *messages*

Sender variables

State of the sender (*S_Ready*, *S_Waiting*, *S_Done*)
VARIABLE *senderState*

All sender variables
$senderVars \triangleq \langle senderState \rangle$

The following variables are all per ledger (functions with domain *Ledger*)

The ledger state (*L_Proposed*, *L_Prepared*, *L_Executed* or *L_Aborted*)
VARIABLE *ledgerState*

All ledger variables
$ledgerVars \triangleq \langle ledgerState \rangle$

Notary variables

State of the notary ($N\_Waiting$, $N\_Committed$, $N\_Aborted$)
VARIABLE $notaryState$

All notary variables
$notaryVars \triangleq \langle notaryState \rangle$

All variables; used for stuttering (asserting state hasn't changed)
$vars \triangleq \langle messages, senderVars, ledgerVars, notaryVars \rangle$

Helpers

Add a set of new messages in transit
$Broadcast(m) \triangleq messages' = messages \oplus SetToBag(m)$

Add a message to the bag of messages
$Send(m) \triangleq Broadcast(\{m\})$

Remove a message from the bag of messages. Used when a process is done processing a message.
$Discard(m) \triangleq messages' = messages \ominus SetToBag(\{m\})$

Respond to a message by sending multiple messages
$ReplyBroadcast(responses, request) \triangleq$
$\quad messages' = messages \ominus SetToBag(\{request\}) \oplus SetToBag(responses)$

Combination of *Send* and *Discard*
$Reply(response, request) \triangleq$
$\quad ReplyBroadcast(\{response\}, request)$

Return the minimum value from a set, or undefined if the set is empty.
$Min(s) \triangleq$ CHOOSE $x \in s : \forall\, y \in s : x \leq y$
Return the maximum value from a set, or undefined if the set is empty.
$Max(s) \triangleq$ CHOOSE $x \in s : \forall\, y \in s : x \geq y$

Is a final ledger state
$IsFinalLedgerState(i) \triangleq i \in \{L\_Executed, L\_Aborted\}$

Sender
$Sender \triangleq Min(Participant)$

Recipient
$Recipient \triangleq Max(Participant)$

Set of connectors
$Connector \triangleq Participant \setminus \{Sender, Recipient\}$

Define type specification for all variables

$$\begin{aligned} TypeOK \triangleq\ & \land\ IsABag(messages) \\ & \land\ senderState \in \{S\_Ready, S\_Waiting, S\_Done\} \\ & \land\ ledgerState\ \in [Ledger \to \{L\_Proposed, L\_Prepared, L\_Executed, L\_Aborted\}] \end{aligned}$$

$$Consistency \triangleq$$
$$\forall\, l1,\, l2 \in Ledger : \neg \land ledgerState[l1] = L\_Aborted \land ledgerState[l2] = L\_Executed$$

$$Inv \triangleq \land TypeOK \land Consistency$$

Define initial values for all variables

$$InitSenderVars \triangleq \land senderState = S\_Ready$$

$$InitLedgerVars \triangleq \land ledgerState = [i \in Ledger \mapsto L\_Proposed]$$

$$InitNotaryVars \triangleq \land notaryState = N\_Waiting$$

$$Init \triangleq \land messages = EmptyBag \land InitSenderVars \land InitLedgerVars \land InitNotaryVars$$

Define state transitions

Participant i starts off the chain

$$Start(i) \triangleq$$
$$\land senderState = S\_Ready$$
$$\land senderState' = S\_Waiting$$
$$\land Send([mtype \mapsto PrepareRequest, msource \mapsto i, mdest \mapsto i+1])$$
$$\land \text{UNCHANGED } \langle ledgerVars,\, notaryVars \rangle$$

Notary times out

$$NotaryTimeout \triangleq$$
$$\land notaryState = N\_Waiting$$
$$\land notaryState' = N\_Aborted$$
$$\land Broadcast(\{[mtype \mapsto AbortRequest, msource \mapsto Notary, mdest \mapsto k] : k \in Ledger\})$$
$$\land \text{UNCHANGED } \langle senderVars,\, ledgerVars \rangle$$

Ledger spontaneously aborts

$$LedgerAbort(l) \triangleq$$
$$\land ledgerState[l] = L\_Proposed$$
$$\land ledgerState' = [ledgerState \text{ EXCEPT } ![l] = L\_Aborted]$$
$$\land Send([mtype \mapsto AbortNotify, msource \mapsto l, mdest \mapsto l-1])$$
$$\land \text{UNCHANGED } \langle senderVars,\, notaryVars \rangle$$

Message handlers

$i =$ recipient, $j =$ sender, $m =$ message

Ledger i receives a Prepare request from process j

$$LedgerHandlePrepareRequest(i,\, j,\, m) \triangleq$$

```
    LET valid ≜  ∧ ledgerState[i] = L_Proposed
                 ∧ j = i − 1
    IN   ∨ ∧ valid
           ∧ ledgerState′ = [ledgerState EXCEPT ![i] = L_Prepared]
           ∧ Reply([mtype   ↦ PrepareNotify,
                    msource ↦ i,
                    mdest   ↦ i + 1], m)
           ∧ UNCHANGED ⟨senderVars, notaryVars⟩
         ∨ ∧ ¬valid
           ∧ Discard(m)
           ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Ledger i receives an Execute request from process j
LedgerHandleExecuteRequest(i, j, m) ≜
    LET valid ≜  ∧ ledgerState[i] = L_Prepared
                 ∧ j = Notary
    IN   ∨ ∧ valid
           ∧ ledgerState′ = [ledgerState EXCEPT ![i] = L_Executed]
           ∧ Reply([mtype   ↦ ExecuteNotify,
                    msource ↦ i,
                    mdest   ↦ i − 1], m)
           ∧ UNCHANGED ⟨senderVars, notaryVars⟩
         ∨ ∧ ¬valid
           ∧ Discard(m)
           ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Ledger i receives an Abort request from process j
LedgerHandleAbortRequest(i, j, m) ≜
    LET valid ≜  ∧ ∨ ledgerState[i] = L_Proposed
                   ∨ ledgerState[i] = L_Prepared
                 ∧ j = Notary
    IN   ∨ ∧ valid
           ∧ ledgerState′ = [ledgerState EXCEPT ![i] = L_Aborted]
           ∧ Reply([mtype   ↦ AbortNotify,
                    msource ↦ i,
                    mdest   ↦ i − 1], m)
           ∧ UNCHANGED ⟨senderVars, notaryVars⟩
         ∨ ∧ ¬valid
           ∧ Discard(m)
           ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Ledger i receives a message
LedgerReceive(i, j, m) ≜
    ∨ ∧ m.mtype = PrepareRequest
      ∧ LedgerHandlePrepareRequest(i, j, m)
    ∨ ∧ m.mtype = ExecuteRequest
      ∧ LedgerHandleExecuteRequest(i, j, m)
    ∨ ∧ m.mtype = AbortRequest
      ∧ LedgerHandleAbortRequest(i, j, m)

 Ledger j notifies sender that the transfer is executed
SenderHandleExecuteNotify(i, j, m) ≜
    ∨ ∧ senderState = S_Waiting
      ∧ senderState′ = S_Done
      ∧ Discard(m)
```

```
        ∧ UNCHANGED ⟨ledgerVars, notaryVars⟩
    ∨ ∧ senderState ≠ S_Waiting
      ∧ Discard(m)
      ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Ledger j notifies sender that the transfer is aborted
SenderHandleAbortNotify(i, j, m) ≜
    LET isSenderWaiting ≜  ∨ senderState = S_Waiting
                           ∨ senderState = S_Ready
    IN    ∨ ∧ isSenderWaiting
            ∧ senderState' = S_Done
            ∧ Discard(m)
            ∧ UNCHANGED ⟨ledgerVars, notaryVars⟩
          ∨ ∧ ¬isSenderWaiting
            ∧ Discard(m)
            ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Sender receives a message
SenderReceive(i, j, m) ≜
    ∨ ∧ m.mtype = ExecuteNotify
      ∧ SenderHandleExecuteNotify(i, j, m)
    ∨ ∧ m.mtype = AbortNotify
      ∧ SenderHandleAbortNotify(i, j, m)

 Ledger j notifies recipient that the transfer is prepared
RecipientHandlePrepareNotify(i, j, m) ≜
    ∧ Reply([mtype    ↦ SubmitReceiptRequest,
             msource  ↦ i,
             mdest    ↦ Notary,
             mreceipt ↦ R_ReceiptSignature], m)
    ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Recipient receives a message
RecipientReceive(i, j, m) ≜
    ∧ m.mtype = PrepareNotify
    ∧ RecipientHandlePrepareNotify(i, j, m)

 Ledger j notifies connector i that the transfer is prepared
ConnectorHandlePrepareNotify(i, j, m) ≜
    ∧ Reply([mtype   ↦ PrepareRequest,
             msource ↦ i,
             mdest   ↦ i + 1], m)
    ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Ledger j notifies connector i that the transfer is executed
ConnectorHandleExecuteNotify(i, j, m) ≜
    ∧ Reply([mtype   ↦ ExecuteRequest,
             msource ↦ i,
             mdest   ↦ i - 1], m)
    ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Ledger j notifies connector i that the transfer is aborted
ConnectorHandleAbortNotify(i, j, m) ≜
    ∧ Discard(m)
    ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Connector receives a message
```

```
ConnectorReceive(i, j, m) ≜
    ∨ ∧ m.mtype = PrepareNotify
      ∧ ConnectorHandlePrepareNotify(i, j, m)
    ∨ ∧ m.mtype = ExecuteNotify
      ∧ ConnectorHandleExecuteNotify(i, j, m)
    ∨ ∧ m.mtype = AbortNotify
      ∧ ConnectorHandleAbortNotify(i, j, m)

 Notary receives a signed receipt
NotaryHandleSubmitReceiptRequest(i, j, m) ≜
    ∨ ∧ m.mreceipt = R_ReceiptSignature
      ∧ notaryState = N_Waiting
      ∧ notaryState' = N_Committed
      ∧ ReplyBroadcast(
          {[mtype   ↦ ExecuteRequest,
            msource ↦ i,
            mdest   ↦ k] : k ∈ Ledger}, m)
      ∧ UNCHANGED ⟨senderVars, ledgerVars⟩
    ∨ ∧ notaryState ≠ N_Waiting
      ∧ Discard(m)
      ∧ UNCHANGED ⟨senderVars, ledgerVars, notaryVars⟩

 Notary receives a message
NotaryReceive(i, j, m) ≜
    ∧ m.mtype = SubmitReceiptRequest
    ∧ NotaryHandleSubmitReceiptRequest(i, j, m)

 Receive a message
Receive(m) ≜
    LET i ≜ m.mdest
        j ≜ m.msource
    IN  ∨ ∧ i ∈ Ledger
          ∧ LedgerReceive(i, j, m)
        ∨ ∧ i = Sender
          ∧ SenderReceive(i, j, m)
        ∨ ∧ i = Recipient
          ∧ RecipientReceive(i, j, m)
        ∨ ∧ i ∈ Connector
          ∧ ConnectorReceive(i, j, m)
        ∨ ∧ i = Notary
          ∧ NotaryReceive(i, j, m)

 End of message handlers
```

```
 Defines how the variables may transition

Termination ≜
    ∧ ∀ l ∈ Ledger : IsFinalLedgerState(ledgerState[l])
    ∧ senderState = S_Done
    ∧ UNCHANGED vars

Next ≜  ∨ Start(Sender)
        ∨ NotaryTimeout
        ∨ ∃ l   ∈ Ledger : LedgerAbort(l)
        ∨ ∃ m ∈ DOMAIN messages : Receive(m)
        ∨ Termination
```

The specification must start with the initial state and transition according to *Next*.

$$Spec \triangleq Init \wedge \Box[Next]_{vars}$$

A.4 Formal Specification: Universal mode

MODULE *Universal*

Formal Specification in TLA$^+$ of the Interledger Protocol *Universal* (*ILP/U*)

```
Modeled after the excellent Raft specification by Diego Ongaro.
  Available at https://github.com/ongardie/raft.tla

  Copyright 2014 Diego Ongaro.
  This work is licensed under the Creative Commons Attribution-4.0
  International License https://creativecommons.org/licenses/by/4.0/
```

EXTENDS *Naturals*, *Sequences*, *FiniteSets*, *Bags*, *TLC*

The set of ledger *IDs*
CONSTANTS *Ledger*

The set of participant *IDs*
CONSTANTS *Participant*

Sender states
CONSTANTS *S_Ready*, *S_ProposalWaiting*, *S_Waiting*, *S_Done*

Connector states
CONSTANTS *C_Ready*, *C_Proposed*

Ledger states
CONSTANTS *L_Proposed*, *L_Prepared*, *L_Executed*, *L_Aborted*

Message types
CONSTANTS *PrepareRequest*, *ExecuteRequest*, *AbortRequest*,
PrepareNotify, *ExecuteNotify*, *AbortNotify*,
SubpaymentProposalRequest, *SubpaymentProposalResponse*

Receipt signature
CONSTANTS *R_ReceiptSignature*

Global variables

Under synchrony we are allowed to have a global clock
VARIABLE *clock*

A bag of records representing requests and responses sent from one process
to another
VARIABLE *messages*

Sender variables

State of the sender (*S_Ready*, *S_Waiting*, *S_Done*)
VARIABLE *senderState*

Whether the sender has received a response from a given connector
VARIABLE *senderProposalResponses*

All sender variables
$senderVars \triangleq \langle senderState, senderProposalResponses \rangle$

Connector variables

State of the connector (C_Ready, $C\_Proposed$)
VARIABLE $connectorState$

All sender variables
$connectorVars \triangleq \langle connectorState \rangle$

The following variables are all per ledger (functions with domain $Ledger$)

The ledger state ($L\_Proposed$, $L\_Prepared$, $L\_Executed$ or $L\_Aborted$)
VARIABLE $ledgerState$

The timeouts for each of the the transfers
VARIABLE $ledgerExpiration$

All ledger variables
$ledgerVars \triangleq \langle ledgerState, ledgerExpiration \rangle$

All variables; used for stuttering (asserting state hasn't changed)
$vars \triangleq \langle clock, messages, senderVars, connectorVars, ledgerVars \rangle$

Helpers

Add a set of new messages in transit
$Broadcast(m) \triangleq messages' = messages \oplus SetToBag(m)$

Add a message to the bag of messages
$Send(m) \triangleq Broadcast(\{m\})$

Remove a message from the bag of messages. Used when a process is done processing a message.
$Discard(m) \triangleq messages' = messages \ominus SetToBag(\{m\})$

Respond to a message by sending multiple messages
$ReplyBroadcast(responses, request) \triangleq$
$\quad messages' = messages \ominus SetToBag(\{request\}) \oplus SetToBag(responses)$

Combination of *Send* and *Discard*
$Reply(response, request) \triangleq$
$\quad ReplyBroadcast(\{response\}, request)$

Return the minimum value from a set, or undefined if the set is empty.
$Min(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s : x \leq y$
Return the maximum value from a set, or undefined if the set is empty.
$Max(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s : x \geq y$

Is a final ledger state
$IsFinalLedgerState(i) \triangleq i \in \{L\_Executed, L\_Aborted\}$

Sender
$Sender \triangleq Min(Participant)$

Recipient
$Recipient \triangleq Max(Participant)$

Set of connectors
$Connector \triangleq Participant \setminus \{Sender, Recipient\}$

The clock value we expect to be at after the proposal phase

$$ClockAfterProposal \triangleq 2 * Cardinality(Connector) + 2$$

The clock value we expect after the preparation phase

$$ClockAfterPrepare \triangleq ClockAfterProposal + 2 * Cardinality(Ledger) + 1$$

The clock value we expect to be at after the execution phase

$$ClockAfterExecution \triangleq ClockAfterPrepare + 2 * Cardinality(Ledger) + 1$$

Define type specification for all variables

$$\begin{aligned} TypeOK \triangleq\ & \land clock \in Nat \\ & \land IsABag(messages) \\ & \land senderState \in \{S\_Ready,\ S\_ProposalWaiting,\ S\_Waiting,\ S\_Done\} \\ & \land senderProposalResponses \in [Connector \to \text{BOOLEAN }] \\ & \land connectorState \in [Connector \to \{C\_Ready,\ C\_Proposed\}] \\ & \land ledgerState \in [Ledger \to \{L\_Proposed,\ L\_Prepared,\ L\_Executed,\ L\_Aborted\}] \\ & \land ledgerExpiration \in [Ledger \to Nat] \end{aligned}$$

$$\begin{aligned} Consistency \triangleq\ & \\ & \forall\, l1,\ l2 \in Ledger : \neg \land ledgerState[l1] = L\_Aborted \\ & \qquad\qquad\qquad\qquad \land ledgerState[l2] = L\_Executed \end{aligned}$$

$$\begin{aligned} Inv \triangleq\ & \land TypeOK \\ & \land Consistency \end{aligned}$$

Define initial values for all variables

$$\begin{aligned} InitSenderVars \triangleq\ & \land senderState = S\_Ready \\ & \land senderProposalResponses = [i \in Connector \mapsto \text{FALSE}] \end{aligned}$$

$$InitConnectorVars \triangleq connectorState = [i \in Connector \mapsto C\_Ready]$$

$$\begin{aligned} InitLedgerVars \triangleq\ & \land ledgerState = [i\ \in Ledger \mapsto L\_Proposed] \\ & \land ledgerExpiration = [i \in Ledger \mapsto ClockAfterExecution - i] \end{aligned}$$

$$\begin{aligned} Init \triangleq\ & \land clock = 0 \\ & \land messages = EmptyBag \\ & \land InitSenderVars \\ & \land InitConnectorVars \\ & \land InitLedgerVars \end{aligned}$$

Define state transitions

Participant i proposes all the subpayments

$$\begin{aligned} StartProposalPhase(i) \triangleq\ & \\ & \land senderState = S\_Ready \\ & \land senderState' = S\_ProposalWaiting \\ & \land Broadcast(\{[\\ & \quad mtype \ \mapsto SubpaymentProposalRequest, \\ & \quad msource \mapsto i, \\ & \quad mdest \ \mapsto k \\ & \quad] : k \in Connector\}) \\ & \land \text{UNCHANGED } \langle ledgerVars,\ connectorVars,\ senderProposalResponses \rangle \end{aligned}$$

Participant i starts off the preparation chain

$StartPreparationPhase(i) \triangleq$
$\land\ senderState = S\_ProposalWaiting$
$\land\ \exists\, p \in \text{DOMAIN}\ senderProposalResponses : senderProposalResponses[p]$
$\land\ senderState' = S\_Waiting$
$\land\ Send([mtype \mapsto PrepareRequest,$
$msource \mapsto i,$
$mdest \mapsto i+1])$
$\land\ \text{UNCHANGED}\ \langle ledgerVars,\ connectorVars,\ senderProposalResponses\rangle$

Ledger spontaneously aborts
$LedgerAbort(l) \triangleq$
$\land\ ledgerState[l] = L\_Proposed$
$\land\ ledgerState' = [ledgerState\ \text{EXCEPT}\ ![l] = L\_Aborted]$
$\land\ Send([mtype \mapsto AbortNotify,$
$msource \mapsto l,$
$mdest \mapsto l-1])$
$\land\ \text{UNCHANGED}\ \langle senderVars,\ connectorVars,\ ledgerExpiration\rangle$

Transfer times out
$LedgerTimeout(l) \triangleq$
$\land\ \neg IsFinalLedgerState(ledgerState[l])$
$\land\ ledgerExpiration[l] \leq clock$
$\land\ ledgerState' = [ledgerState\ \text{EXCEPT}\ ![l] = L\_Aborted]$
$\land\ Send([mtype \mapsto AbortNotify,$
$msource \mapsto l,$
$mdest \mapsto l-1])$
$\land\ \text{UNCHANGED}\ \langle senderVars,\ connectorVars,\ ledgerExpiration\rangle$

If no messages are in flight and the sender isn't doing anything, advance the clock
$NothingHappens \triangleq$
$\land\ clock \leq Max(\{ledgerExpiration[x] : x \in Ledger\})$
$\land\ BagCardinality(messages) = 0$
$\land\ senderState \neq S\_Ready$
$\land\ \lor\ senderState \neq S\_ProposalWaiting$
$\lor\ \forall\, p \in \text{DOMAIN}\ senderProposalResponses : \neg senderProposalResponses[p]$
$\land\ \text{UNCHANGED}\ \langle messages,\ senderVars,\ connectorVars,\ ledgerVars\rangle$

Message handlers
$i =$ recipient, $j =$ sender, $m =$ message

Ledger i receives a Prepare request from participant j
$LedgerHandlePrepareRequest(i,\ j,\ m) \triangleq$
LET $valid \triangleq \land\ ledgerState[i] = L\_Proposed$
$\land\ j = i - 1$
IN $\lor\ \land\ valid$
$\land\ i \in Ledger$
$\land\ ledgerState' = [ledgerState\ \text{EXCEPT}\ ![i] = L\_Prepared]$
$\land\ Reply([mtype \mapsto PrepareNotify,$
$msource \mapsto i,$
$mdest \mapsto i+1],\ m)$
$\land\ \text{UNCHANGED}\ \langle senderVars,\ connectorVars,\ ledgerExpiration\rangle$
$\lor\ \land\ \neg valid$
$\land\ i \in Ledger$
$\land\ Discard(m)$

```
        ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Ledger i receives an Execute request from process j
LedgerHandleExecuteRequest(i, j, m) ≜
    LET valid ≜  ∧ ledgerState[i] = L_Prepared
                 ∧ ledgerExpiration[i] > clock
                 ∧ m.mreceipt = R_ReceiptSignature
    IN   ∨ ∧ valid
           ∧ ledgerState' = [ledgerState EXCEPT ![i] = L_Executed]
           ∧ Reply([mtype     ↦ ExecuteNotify,
                    msource   ↦ i,
                    mdest     ↦ i − 1,
                    mreceipt  ↦ m.mreceipt], m)
           ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerExpiration⟩
         ∨ ∧ ¬valid
           ∧ Discard(m)
           ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Ledger i receives a message
LedgerReceive(i, j, m) ≜
    ∨ ∧ m.mtype = PrepareRequest
      ∧ LedgerHandlePrepareRequest(i, j, m)
    ∨ ∧ m.mtype = ExecuteRequest
      ∧ LedgerHandleExecuteRequest(i, j, m)

 Sender receives a SubpaymentProposal request
SenderHandleSubpaymentProposalResponse(i, j, m) ≜
    ∧ i = Sender
    ∧ senderProposalResponses' = [senderProposalResponses EXCEPT ![j] = TRUE]
    ∧ Discard(m)
    ∧ UNCHANGED ⟨connectorVars, ledgerVars, senderState⟩

 Ledger j notifies sender that the transfer is executed
SenderHandleExecuteNotify(i, j, m) ≜
    ∨ ∧ senderState = S_Waiting
      ∧ senderState' = S_Done
      ∧ Discard(m)
      ∧ UNCHANGED ⟨ledgerVars, connectorVars, senderProposalResponses⟩
    ∨ ∧ senderState ≠ S_Waiting
      ∧ Discard(m)
      ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Ledger j notifies sender that the transfer is aborted
SenderHandleAbortNotify(i, j, m) ≜
    LET isSenderWaiting ≜ senderState ∈ {S_ProposalWaiting, S_Waiting, S_Ready}
    IN   ∨ ∧ isSenderWaiting
           ∧ senderState' = S_Done
           ∧ Discard(m)
           ∧ UNCHANGED ⟨ledgerVars, connectorVars, senderProposalResponses⟩
         ∨ ∧ ¬isSenderWaiting
           ∧ Discard(m)
           ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Sender receives a message
SenderReceive(i, j, m) ≜
    ∨ ∧ m.mtype = SubpaymentProposalResponse
```

```
        ∧ SenderHandleSubpaymentProposalResponse(i, j, m)
    ∨ ∧ m.mtype = ExecuteNotify
        ∧ SenderHandleExecuteNotify(i, j, m)
    ∨ ∧ m.mtype = AbortNotify
        ∧ SenderHandleAbortNotify(i, j, m)

 Ledger j notifies recipient that the transfer is prepared
RecipientHandlePrepareNotify(i, j, m) ≜
    ∨ ∧ Reply([mtype    ↦ ExecuteRequest,
               msource  ↦ i,
               mdest    ↦ i − 1,
               mreceipt ↦ R_ReceiptSignature], m)
        ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Recipient receives a message
RecipientReceive(i, j, m) ≜
    ∨ ∧ m.mtype = PrepareNotify
        ∧ RecipientHandlePrepareNotify(i, j, m)

 Connector i receives a SubpaymentProposal request
ConnectorHandleSubpaymentProposalRequest(i, j, m) ≜
    ∧ connectorState′ = [connectorState EXCEPT ![i] = C_Proposed]
    ∧ Reply([mtype    ↦ SubpaymentProposalResponse,
             msource  ↦ i,
             mdest    ↦ j], m)
    ∧ UNCHANGED ⟨senderVars, ledgerVars⟩

 Ledger j notifies connector i that the transfer is prepared
ConnectorHandlePrepareNotify(i, j, m) ≜
    ∨ ∧ Reply([mtype    ↦ PrepareRequest,
               msource ↦ i,
               mdest    ↦ i + 1], m)
        ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Ledger j notifies connector i that the transfer is executed
ConnectorHandleExecuteNotify(i, j, m) ≜
    ∧ Reply([mtype    ↦ ExecuteRequest,
             msource  ↦ i,
             mdest    ↦ i − 1,
             mreceipt ↦ m.mreceipt], m)
    ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Ledger j notifies connector i that the transfer is aborted
ConnectorHandleAbortNotify(i, j, m) ≜
    ∧ Discard(m)
    ∧ UNCHANGED ⟨senderVars, connectorVars, ledgerVars⟩

 Connector receives a message
ConnectorReceive(i, j, m) ≜
    ∨ ∧ m.mtype = SubpaymentProposalRequest
        ∧ ConnectorHandleSubpaymentProposalRequest(i, j, m)
    ∨ ∧ m.mtype = PrepareNotify
        ∧ ConnectorHandlePrepareNotify(i, j, m)
    ∨ ∧ m.mtype = ExecuteNotify
        ∧ ConnectorHandleExecuteNotify(i, j, m)
    ∨ ∧ m.mtype = AbortNotify
```

$\land$ *ConnectorHandleAbortNotify*$(i,\ j,\ m)$

Receive a message

Receive(m) $\triangleq$
LET $i \triangleq m.mdest$
$j \triangleq m.msource$
IN $\lor \land i \in Ledger$
$\land$ *LedgerReceive*$(i,\ j,\ m)$
$\lor \land i = Sender$
$\land$ *SenderReceive*$(i,\ j,\ m)$
$\lor \land i = Recipient$
$\land$ *RecipientReceive*$(i,\ j,\ m)$
$\lor \land i \in Connector$
$\land$ *ConnectorReceive*$(i,\ j,\ m)$

End of message handlers

Defines how the variables may transition

Termination $\triangleq$
$\land \forall\, l \in Ledger : IsFinalLedgerState(ledgerState[l])$
$\land\ senderState = S\_Done$
$\land$ UNCHANGED *vars*

Next $\triangleq$ $\lor \land \lor$ *StartProposalPhase*$(Sender)$
$\lor$ *StartPreparationPhase*$(Sender)$
$\lor \exists\, l \in Ledger : LedgerAbort(l)$
$\lor \exists\, l \in Ledger : LedgerTimeout(l)$
$\lor \exists\, m \in$ DOMAIN *messages* $: Receive(m)$
$\lor$ *NothingHappens*
$\land\ clock' = clock + 1$
$\lor$ *Termination*

The specification must start with the initial state and transition according to *Next*.

Spec $\triangleq$ $Init \land \Box[Next]_{vars}$
