pay=(type, rnd, aux)

$$\begin{cases} type=dirp,\ aux=(pr, pe, amt) \\ type=conp,\ aux=(pr, pe, amt, h, time) \\ type=fulp.\ aux=(cmd, cpay). \end{cases}$$

$(Input_j=(pay_j, bal_j, ConPay_j), \sigma_j)$

$state=(rnd, \{Input_j, \sigma_j\}_j, F)$

$state_r=(r, \{Input_{r_j}, \sigma_{r_j}\}_j, F_r)$

性质:

① state中,包含的各节点余额、各条件支付的金额与手续费之和与通道内总金额相等.

② state中包含的每个条件支付要同时出现在交易双方的条件支付集合中.

③ state中,各节点的轮次都要不大于state的轮次,并且不大于交易对方的轮次,若轮次相等,则两人包含的交易相同

④ 对于$state_r$和$state_{r'}$, $r'>r$. 则对节点P_j来说,在$state_r$中的轮次记为r_j,在$state_{r'}$中的轮次记为r_j'.则 $\begin{cases} r_j'=r_j \\ r_j'>r. \end{cases}$

在 $state_r \to state_{r'}$, $r' > r+1$ 时.

若包含错误签名的节点的状态. 可直接被验证.

若包含的都是节点自己签名的状态. 只从诚实节点 P_j 的角度看.

(1) 假设节点 P_j 没有发送 $r+1$ 轮的状态. 则 custodian 将操作 P_j 的轮次 $r_c \le r$.

假设在 $state_r$ 中包含的轮次为 r_j. 在 $state_{r'}$ 中包含的轮次为 r_j'.

- 若正常更新. 则应该包含 r_j. 即 $r_j' = r_j$.
- 若不正常. 则 $r_j' \neq r_j$ 且 $r_j' < r$. 则与④相悖.

(2) 假设 P_j 发送了 $r+1$ 轮的状态. 只分为两种情况. 包含与不包含.

若包含. 则相当于认可这笔交易. 此时对节点更新后的状态也应该被包含.

如果节点诚实, 那么更新到 $r+1$ 轮. 交易相同.

如果节点与 custodian 合谋, 则可能会销毁 $r+1$ 的状态. 以按需减少节点的利益

体现在余额上. 可能会自己销. 但这不会影响诚实节点的余额. 则与总金额不等. 违反①

在条件支付上, 可能会单方面将 cpay 完成. 违反③.

若不包含. 则与(1)相同.

因

AuditSin(state$_r$).

if(!VerSig() $\vee$!CheckSum() $\vee$!CheckConpay() $\vee$!checkRound()).
    return false
else return true.

VerSig(state$_r$).
  parse State$_r$ as $(r, \lceil Input_j.\sigma_j \rfloor_j, F_r)$
  if $\exists j$, $!Verify_j(Input_j.\sigma_j)$, return false.
  else return true.

CheckSum(state$_r$)
  parse State$_r$ as $(r, \lceil Input_j.\sigma_j \rfloor_j, F_r)$
  parse Input$_j$ as $(pay_{j_i}, bal_j, ConPay_j)$.
  if $(F_r + \sum_j (bal_j + \frac{1}{2}\sum cpay.amt, cpay \in ConPay_j) \neq totalValue)$.
    return false
  else return true.

CheckConpay(state$_r$).
  parse State$_r$ as $(r, \lceil Input_j.\sigma_j \rfloor_j, F_r)$
  parse Input$_j$ as $(pay_{j_i}, bal_j, ConPay_j)$.
  if $\exists cpay$, set $P_i := cpay.pr$, $P_j := cpay.pe$. $cpay \notin (ConPay_i \cap ConPay_j)$.
    return false.
  else return true.

CheckRound(state$_r$).
  parse State$_r$ as $(r, \lceil Input_j.\sigma_j \rfloor_j, F_r)$
  parse Input$_j$ as $(pay_{j_i}, bal_j, ConPay_j)$.
  if $\exists pay_j$, $P_i = pay_j.other(P_j)$.
    $(pay_j.rnd > r) \vee (pay_i.rnd > r) \vee (pay_i.rnd < pay_j.rnd)$
    $\vee [(pay_j.rnd == pay_i.round) \wedge (pay_j \neq pay_i)]$
    return false
  else return true.

AuditDou(state$_r$, state$_{r'}$), $r' \geq r$.
  if $[(r = r') \wedge (state_r \neq state_{r'})] \vee [(r' = r+1) \wedge (state_{r'} \neq Update(state_r))]$
    $\vee [(r' > r+1) \wedge [\exists j, (r_j \neq r_j') \wedge (r_j' \leq r)]]$, return false.
  else return true

链上合约中的判断.

Audit(proof)
parse proof as $(state_{r_1}, \sigma_{r_1}, state_{r_2}, \sigma_{r_2})$.
for r in $[r_1, r_2]$
  if $!VerSig_{cus}(state_r, \sigma_r) \wedge (state_r \neq BestState)$, $state_r := \perp$.
  if $VerSig_{cus}(state_r, \sigma_r) \wedge !AuditSin(state_r)$, punish()

if $(state_{r_1} \neq \perp) \wedge (state_{r_2} \neq \perp) \wedge !AuditDou(state_{r_1}, state_{r_2})$.
  punish().

节点离开通道，发送最新状态.

Depart(state, σ). from P_j at T:
  if $DP = \phi$, set deadline := $T + \Delta$.
  $DP := DP \cup \{P_j\}$.
  Record(state, σ).
  emit EventDepart(state).

Record($state_r$, σ). at T.
  assert $T <$ deadline.
  Audit($state_r$, σ, BestState, ⊥)
  if ($r >$ BestRound), BestState := $state_r$, BestRound := r.

Resolve( ) at T:
  assert $DP \neq \phi \wedge T >$ deadline.
  BestState = UpdateCon(BestState).
  parse BestState as $(r, \{Input_j, \perp\}_j, F)$.
  if $(cus \in DP) \wedge (\forall j.\ ConPay_j = \phi)$,
    send bal_j to P_j
    send $F + G$ to cus.
    emit EventClose
  else for each $P_j \in DP \wedge P_j \neq cus$,
      if $bal_j = \phi$, send bal_j to P_j.
      $P = P \setminus \{P_j\}$.
      totalValue −= bal_j.
    BestState := $(r+1, \{Input_j, \perp_j\}_{P_j \in N}, F)$.
    BestRound := $r+1$
    DP := ϕ.
    emit EventResolve(BestState).

UpdateCon(state). at T:
  parse state as $(r, \{Input_j, \sigma_j\}_j, F)$.
  parse each cpay as (conp, rnd, pr, pe, amt, h, time)
  if (time > T)
    $ConPay_{pr} = ConPay_{pr} \setminus \{cpay\}$. $ConPay_{pe} = ConPay_{pe} \setminus \{cpay\}$
    if PM(h, time), bal_{pe} += amt,
    else bal_{pr} += amt.
  return $(r, \{Input_j, \perp\}_j, F)$.

Punish( )
  c = (totalValue + G)/(|P|−1)
  send $c to parties in P except cus.
  emit EventClose

节点离开通道，发送最新状态.

Depart(state, σ). from P_j at T:
  if DP = ϕ, set deadline := T + Δ.
  DP := DP ∪ {P_j}.
  Record(state, σ).
  emit EventDepart(state).

Record(state$_r$, σ). at T.
  assert T < deadline.
  Audit(state$_r$, σ, BestState, ⊥)
  if (r > BestRound), BestState := state$_r$, BestRound := r.

Resolve( ) at T:
  assert DP ≠ ϕ ∧ T > deadline.
  BestState = UpdateCon(BestState).
  parse BestState as (r, {Input$_j$, ⊥}$_j$, F).
  if (cus ∈ DP) ∧ (∀j. ConPay$_j$ = ϕ),
    send bal$_j$ to P_j
    send F + G to cus.
    emit EventClose
  else for each P_j ∈ DP ∧ P_j ≠ cus,
    if bal$_j$ = ϕ, send bal$_j$ to P_j.
    P = P \ {P_j}.
    totalValue −= bal$_j$.
  BestState := (r+1, {Input$_j$, ⊥}$_{j, P_j \in N}$, F).
  BestRound := r+1
  DP := ϕ.
  emit EventResolve(BestState).

UpdateCon(state). at T:
  parse state as (r, {Input$_j$, σ$_j$}$_j$, F).
  parse each cpay as (conp, rnd, pr, pe, amt, h, time)
  if (time > T)
    ConPay$_{pr}$ = ConPay$_{pr}$ \ {cpay}. ConPay$_{pe}$ = ConPay$_{pe}$ \ {cpay}
    if pre(h, time), bal$_{pe}$ += amt.
    else bal$_{pr}$ += amt.
  return (r, {Input$_j$, ⊥}$_j$, F).

Punish( )
  c = (totalValue + G)/(|P| − 1)
  send $c to parties in P except cus.
  emit EventClose

Contract MPC.

Initialize BestState := (0, {(ϕ, deposit$_j$, ϕ), ⊥}$_j$, 0)
  BestRound := 0.
  P, totalValue = Σ$_j$ deposit$_j$.

Depart( ) ← triggered
Record( ) ← triggered. 调用.
Resolve( ) ← triggered.
punish( ) ← 只能合约内部调用.
UpdateCon( ) ← 只能合约内部调用.
Audit( ) ← trigged 或内部调用.
  {AuditSig
  AuditDou. 被内部调用.

pay=(type, rnd, aux)

- type=dirp, aux=(pr, pe, amt)
- type=conp, aux=(pr, pe, amt, h, time)
- type=fulp, aux=(cmd, cpay).

(Input_j=(pay_j, bal_j, ConPay_j), σ_j)

state=(rnd, {Input_j, σ_j}_j, F)

state_r=(r, {Input_{r_j}, σ_{r_j}}_j, F_r)

性质：

① state中，包含的各节点余额、各条件支付的金额与手续费之和与通道内总金额相等.

② State中包含的每个条件支付要同时出现在交易双方的条件支付集合中.

③ State中，各节点的轮次都要不大于state的轮次，并且不大于交易对方的轮次，若轮次相等，则两个包含的交易相同

④ 对于state_r 和 state_{r'}，r'>r. 则对节点P_j来说，在State_r中的轮次记为 r_j，在state_{r'}中的轮次记为 r_j'. 则 r_j ≤ r_j', 且 r_j' > r.

在 state_r → state_{r'}, r'>r+1 时.

若包含错误签名的节点的状态，可直接裁决验证.

若包含的都是节点自己签名的状态，只可从诚实节点 P_j 的角度看.

(1) 假设节点 P_j 没有发送 r+1 轮的状态，则 custodian 将强制 P_j 的轮次 r_c ≤ r.

假设在 state_r 中包含的轮次为 r_j，在 state_{r'} 中包含的轮次为 r_j'.

- 若正常更新，则应该包含 r_j，即 r_j' = r_j.
- 若不正常，则 r_j' ≠ r_j 且 r_j' < r，则与④相悖.

(2) 假设 P_j 发送了 r+1 轮的状态，只会为两种情况，包含与不包含.

若包含，则相当于认可这笔交易，此时对应更新后的状态也应该被包含.

如果拒绝或超时，那么更新到 r+1 轮，交易相同.

如果双方与 Cus 合谋，则可能会创造 r+1 的状态，以换回诚实节点的利益

体现在余额上，可能会自己减，但这不会影响诚实节点的余额，则与总金额不等，违反①

在条件支付上，可能会单方面将 cpay 完成，违反③.

若不包含，则与(1)相同.

```
AuditDou(state_r, state_{r'}), r'≥r.
  if [(r=r') ∧ (state_r ≠ state_{r'})] ∨ [(r'=r+1) ∧ (state_{r'} ≠ Update(state_r))]
     ∨ [(r'>r+1) ∧ [∃j, (r_j ≠ r_j') ∧ (r_j' ≤ r)]], return false.
  else return true
```

链上合约中的判断.

```
Audit(proof)
parse proof as (state_{r_1}, σ_{r_1}, state_{r_2}, σ_{r_2}).
for r in [r_1, r_2]
    if !VerSig_cus(state_r, σ_r) ∧ (state_r ≠ BestState), state_r := ⊥.
    if VerSig_cus(state_r, σ_r) ∧ !AuditSin(state_r), punish()
if (state_{r_1} ≠ ⊥) ∧ (state_{r_2} ≠ ⊥) ∧ !AuditDou(state_{r_1}, state_{r_2}).
   punish().
```

因此，AuditSin(state_r).

```
if(!VerSig() ∨ !CheckSum() ∨ !CheckConpay() ∨ !CheckRound()).
   return false
else return true.

VerSig(state_r).
  parse State_r as (r, {Input_j, σ_j}_j, F_r)
  if ∃j, !Verify_j(Input_j, σ_j), return false.
  else return true.
CheckSum(state_r)
  parse State_r as (r, {Input_j, σ_j}_j, F_r)
  parse Input_j as (pay_j, bal_j, ConPay_j).
  if (F_r + Σ_j (bal_j + Σ_{i∈cpay.amt, cpay∈ConPay_j} cpay.amt) ≠ totalValue).
      return false
  else return true.
CheckConpay(state_r).
  parse State_r as (r, {Input_j, σ_j}_j, F_r)
  parse Input_j as (pay_j, bal_j, ConPay_j).
  if ∃cpay, set P_i := cpay.pr, P_j := cpay.pe. cpay ∉ (ConPay_i ∩ ConPay_j).
      return false.
  else return true.
CheckRound(state_r).
  parse State_r as (r, {Input_j, σ_j}_j, F_r)
  parse Input_j as (pay_j, bal_j, ConPay_j).
  if ∃pay_j, P_i = pay_j.other(P_j).
    (pay_j.rnd > r) ∨ (pay_i.rnd > r) ∨ (pay_i.rnd < pay_j.rnd)
     ∨ [(pay_j.rnd == pay_i.round) ∧ (pay_j ≠ pay_i)]
     return false
  else return true.
```