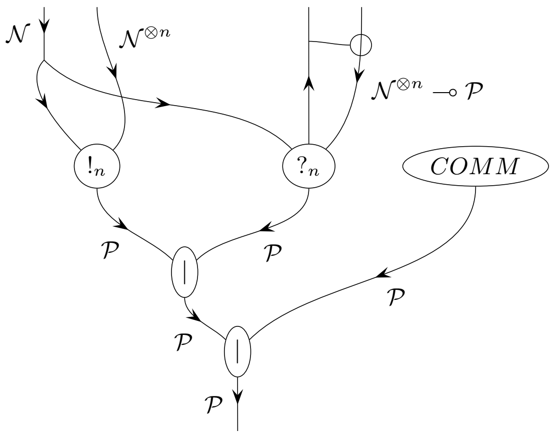
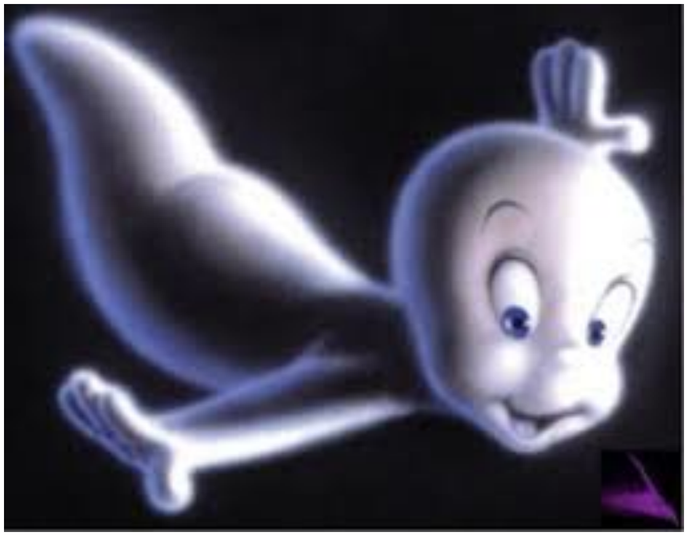
Casper Formalized Pt I

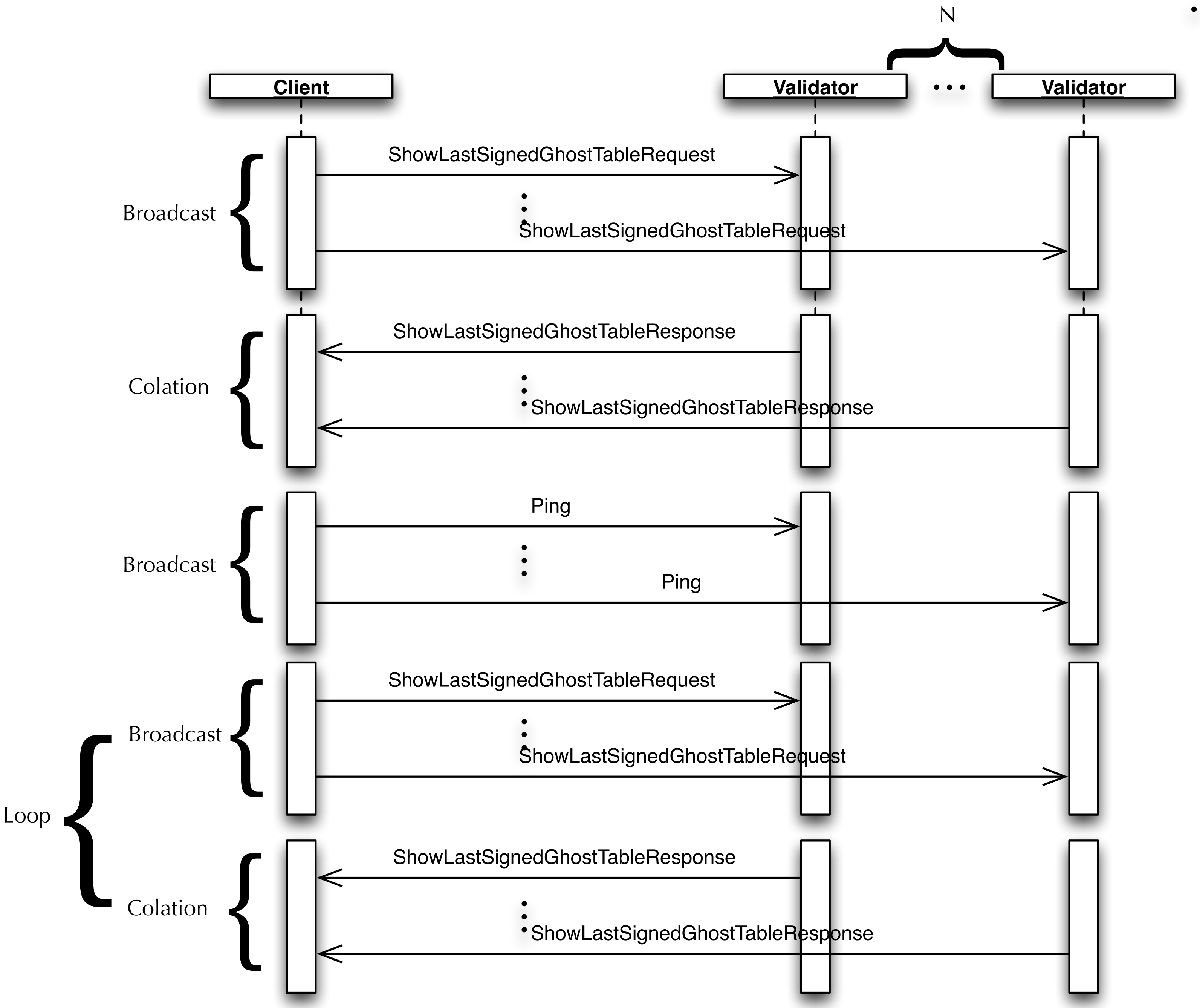
Applying π-calculus to modeling the Casper protocol



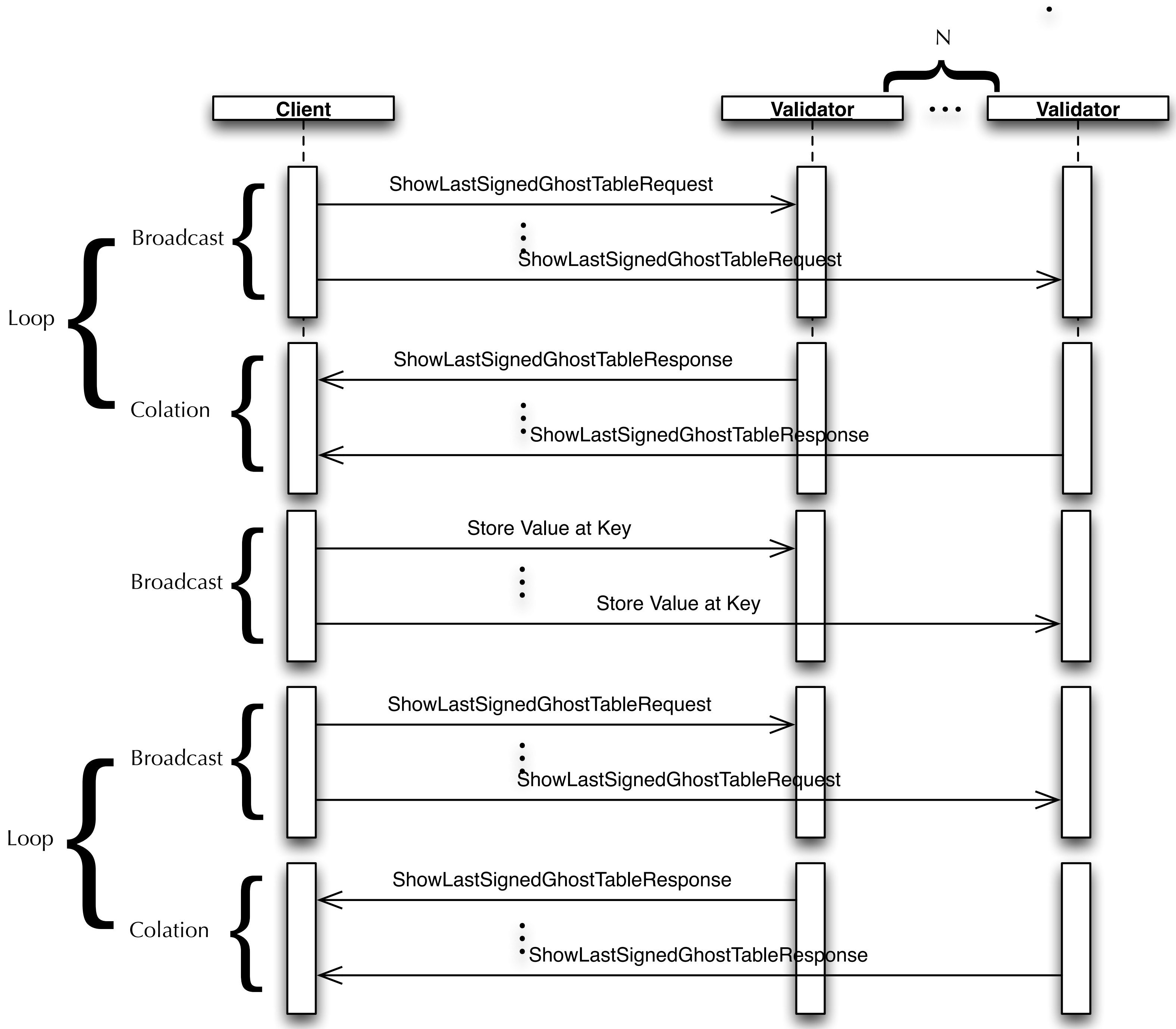


Vlad Zamfir, L.G. Meredith

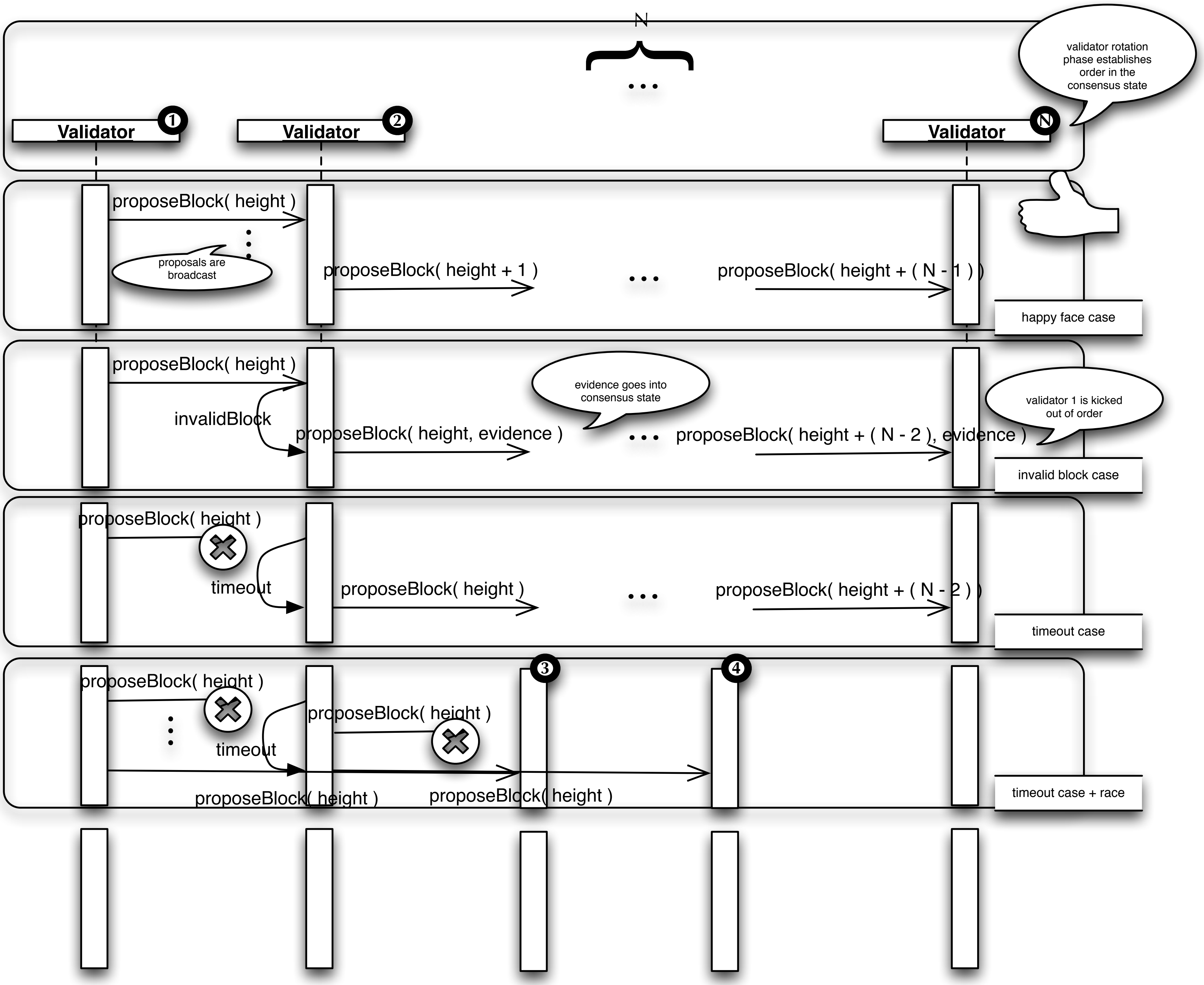
Client <-> Validator interactions



N
Client
Validator
Validator
Broadcast
ShowLastSignedGhostTableRequest
ShowLastSignedGhostTableRequest
Colation
ShowLastSignedGhostTableResponse
ShowLastSignedGhostTableResponse
Broadcast
Ping
Ping
Loop
Broadcast
ShowLastSignedGhostTableRequest
ShowLastSignedGhostTableRequest
Colation
ShowLastSignedGhostTableResponse
ShowLastSignedGhostTableResponse

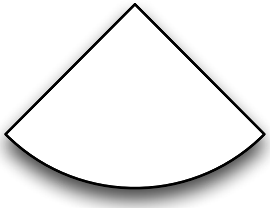

N
Client
Validator
Validator
Loop
Broadcast
ShowLastSignedGhostTableRequest
ShowLastSignedGhostTableRequest
Colation
ShowLastSignedGhostTableResponse
ShowLastSignedGhostTableResponse
Broadcast
Store Value at Key
Store Value at Key
Loop
Broadcast
ShowLastSignedGhostTableRequest
ShowLastSignedGhostTableRequest
Colation
ShowLastSignedGhostTableResponse
ShowLastSignedGhostTableResponse

Consensus: Validator <-> Validator interactions

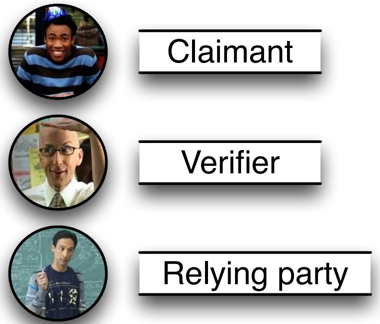


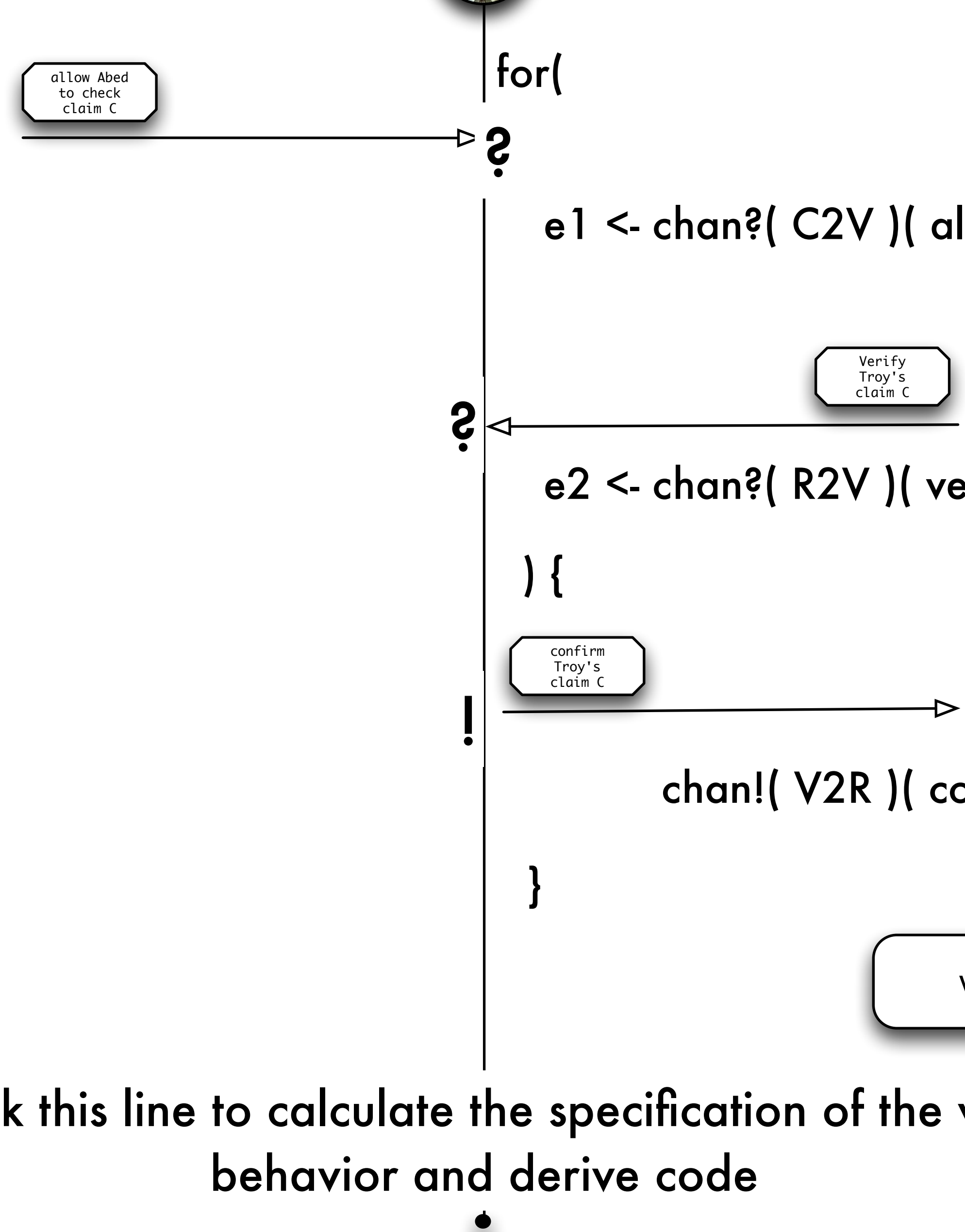
N
...
Validator
1
Validator
2
Validator
N
validator rotation phase establishes order in the consensus state
proposeBlock(height)
proposals are broadcast
proposeBlock(height + 1)
...
proposeBlock(height + (N - 1))
happy face case
proposeBlock(height)
invalidBlock
evidence goes into consensus state
proposeBlock(height, evidence)
...
proposeBlock(height + (N - 2), evidence)
validator 1 is kicked out of order
invalid block case
proposeBlock(height)
timeout
proposeBlock(height)
...
proposeBlock(height + (N - 2))
timeout case
3
4
proposeBlock(height)
timeout
proposeBlock(height)
proposeBlock(height)
proposeBlock(height)
timeout case + race

How to turn interaction diagrams into π-calculus specs



programming model

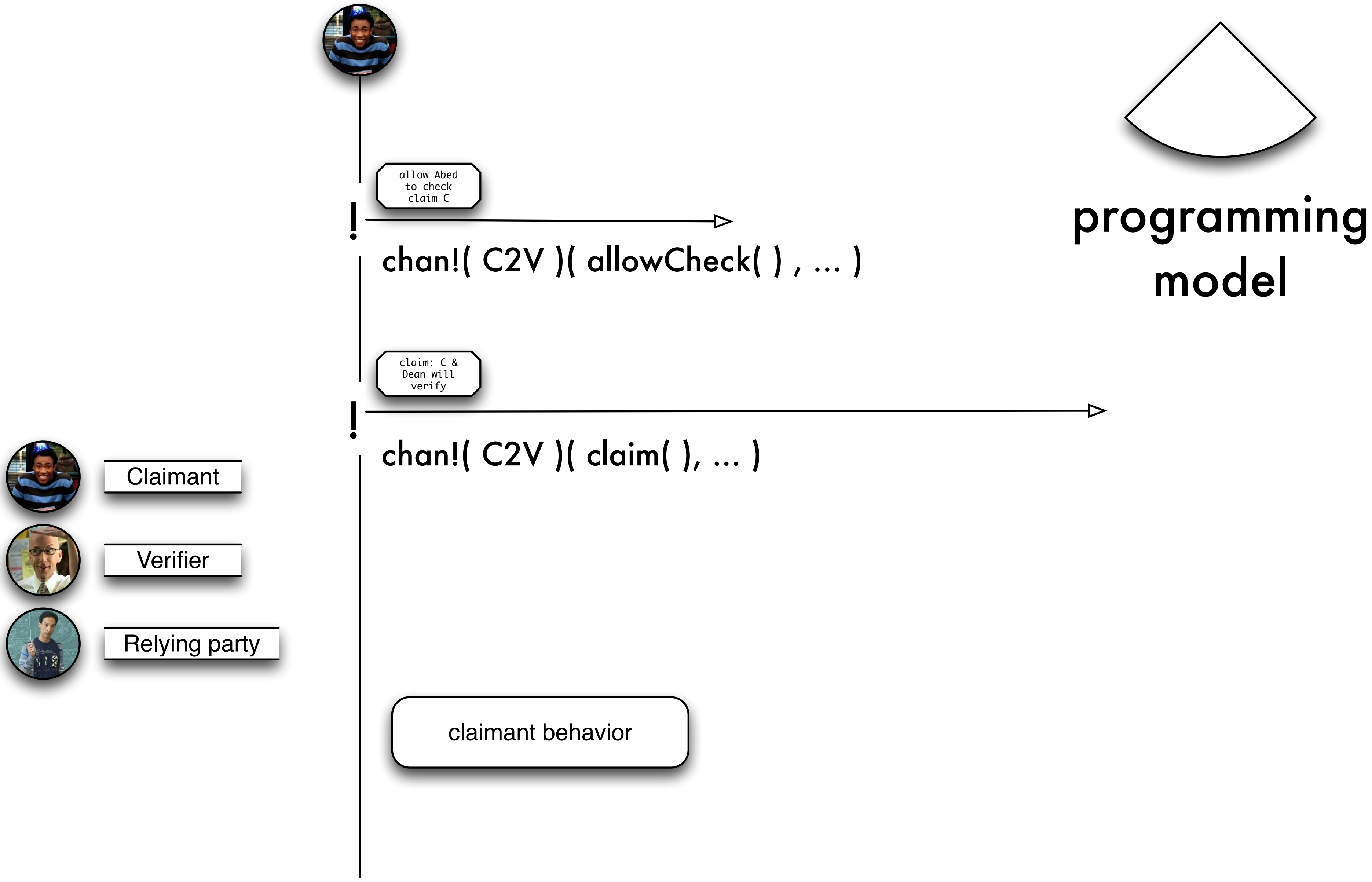




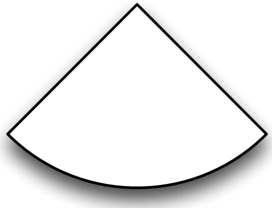

```
for(
  e1 <- chan?( C2V )( allowCheck( ) ) );
  e2 <- chan?( R2V )( verify( ) )
) {
    chan!( V2R )( confirm( ) )
}
```

verifier behavior

Just walk this line to calculate the specification of the verifier's behavior and derive code



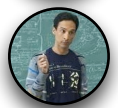
and this line to calculate the specification of the claimant's behavior and derive code

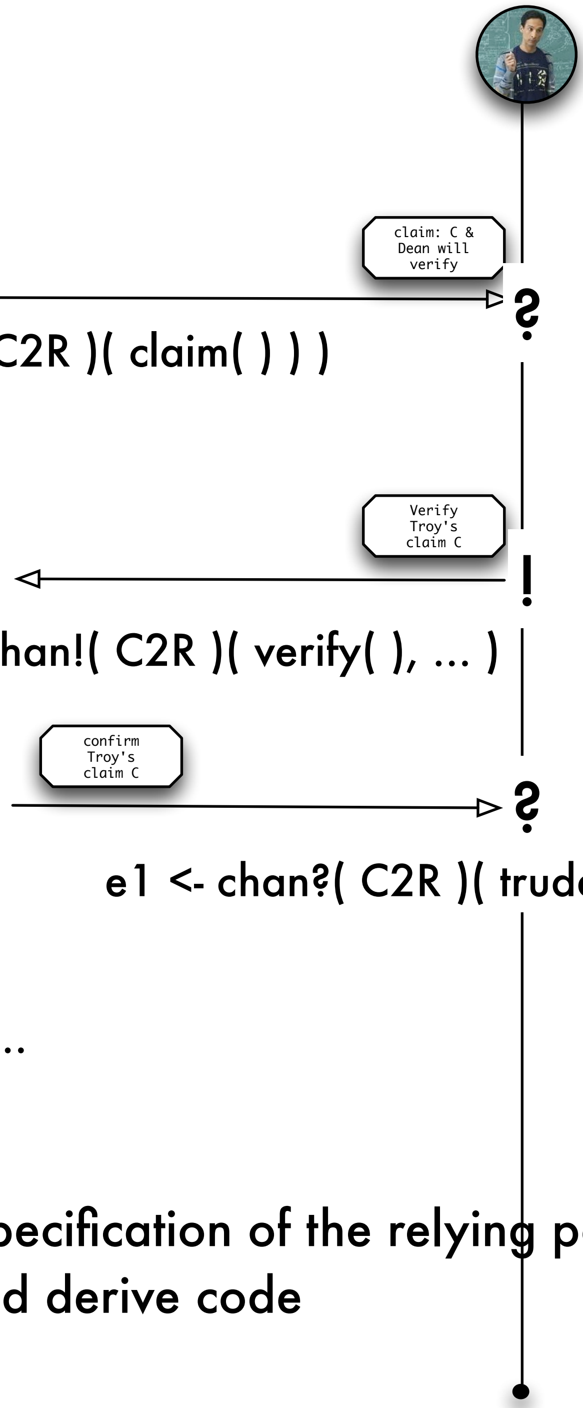


programming model


Claimant


Verifier


Relying party




```
for(
    e1 <- chan?( C2R )( claim( ) ) )
) {
    chan!( C2R )( verify( ), ... )
    for(
        e1 <- chan?( C2R )( trudat( ) ) )
    ) {
        ...
    }
}
```

relying party behavior

and this line to calculate the specification of the relying party's behavior and derive code

How to turn π-calculus specs into scala code

π-calculus

```
P,Q  ::=  0
          a![ t ]Q
          a?( t )P
           P | Q
          (new a)P
          ( def X( t ) = P )[ u ]
           X[ t ]

t,u   ::=   g
          var
           functor( t* )
g     ::=   bool  |  int  |  string  |  double
var   ::=   _   |  upperCaseIdent
functor  ::=  l  lowerCaseIdent
```

A spec for the applied π-calculus

Structural congruence

The *structural congruence* of processes, noted $\equiv$, is the least congruence containing α-equivalence, $\equiv_\alpha$, making $(P, |, 0)$ into commutative monoids and satisfying

$$(\mathsf{new}\ x)(\mathsf{new}\ x)P \equiv (\mathsf{new}\ x)P$$

$$(\mathsf{new}\ x)(\mathsf{new}\ y)P \equiv (\mathsf{new}\ y)(\mathsf{new}\ x)P$$

$$((\mathsf{new}\ x)P) \mid Q \equiv (\mathsf{new}\ x)(P \mid Q)$$

$$\frac{\text{unify}(\ t,\ u,\ s\ )}{a?(\ t\ )P \mid a![\ u\ ]Q \text{ -> } Ps \mid Qs}$$

$$\frac{P \text{ -> } P'}{P \mid Q \text{ -> } P' \mid Q}$$

$$\frac{P \text{ -> } P'}{(\text{new } a)P \text{ -> } (\text{new } a)P'}$$

$$\frac{P \equiv P' \qquad P' \text{ -> } Q' \qquad Q' \equiv Q}{P \text{ -> } Q}$$

A spec for the applied π-calculus

P,Q ::= 0

```
{ }
```

a![v1, …, vn]

```
[| a |]( m ) ![ [| v1 |]( m ), …, [| vn |]( m ) ]
```

a?(x1, …, xn)P

```
for( [ x1, …, xn ] <- [| a |]( m ) ){
    [| P |]( m )( x1, …, xn )
}
```

P | Q

```
spawn{ [| P |]( m ) };spawn{ [| Q |]( m ) }
```

(new a)P

```
{ val q = new Queue(); [| P |]( m[ a <- q ] ) }
```

(def X(x1, …, xn) = P)[v1, …, vn]

```
object X {
   def apply( x1, …, xn ) = {
      [| P |]( m ) ( x1, …, xn )
   }
}
```

X[v1, …, vn]

```
X( [| v1 |]( m ), …, [| vn |]( m ) )
```

```
[| - |]( - ) : ( π-calculus, Map[Symbol,Queue] ) -> Scala
```

$$
\begin{tikzcd}
 & P \arrow[dl] \arrow[dr] & \\
P' \arrow[dr] & & P'' \arrow[dl] \\
 & Q &
\end{tikzcd}
$$

Confluence

lambda calculus is confluent

π-calculus is not confluent

a![u1]Q1 | a?(t)P | a![u2]Q2

typical race condition