Hello, would you like to hear a TCP joke?

Yes, I'd like to hear a TCP joke.

OK, I'll tell you a TCP joke.

OK, I'll hear a TCP joke.

Are you ready to hear a TCP joke?

Yes, I am ready to hear a TCP joke.

OK, I'm about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline.

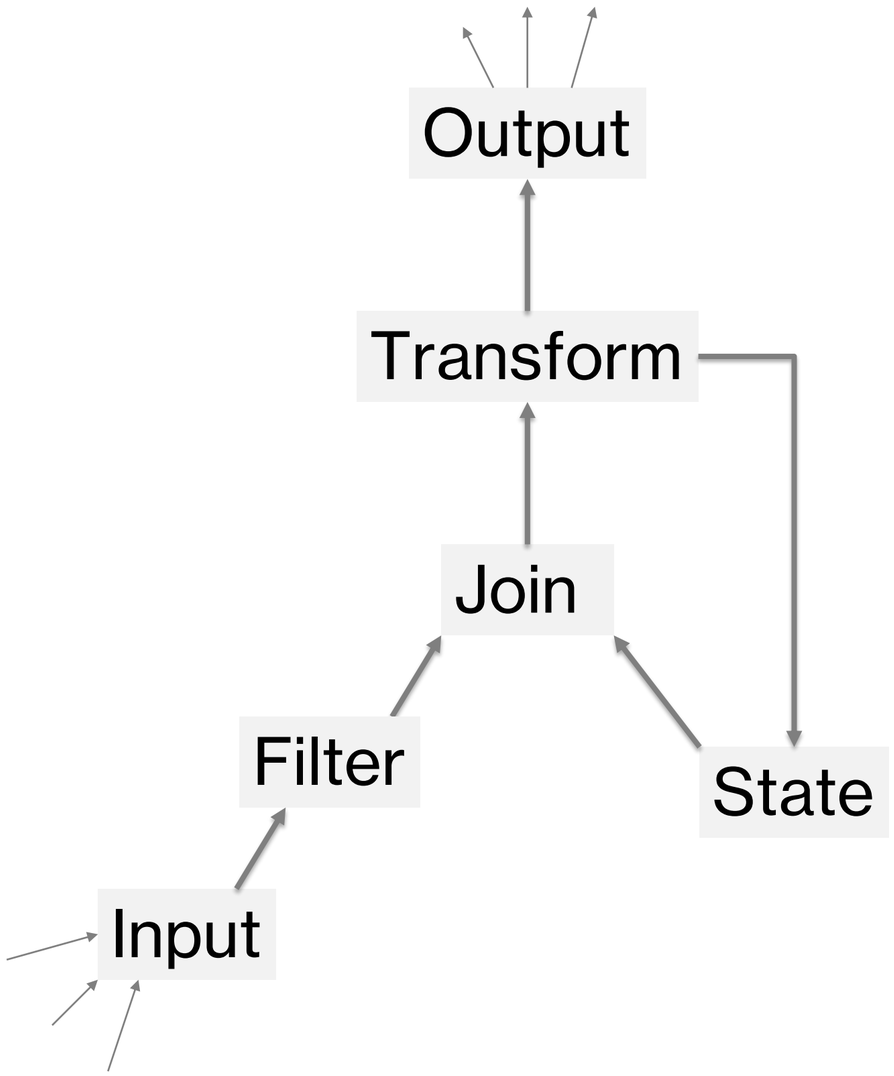
OK, I'm ready to hear the TCP joke that will last 10 seconds, has two characters, does not have a setting and will end with a punchline.

I'm sorry, your connection has timed out... ...Hello, would you like to hear a TCP joke?

Distributed systems

1: Processes

The generalized process




```
output = fn1(input, state)

state  = fn2(input, state)
```

Observations I: internal

Two flows: Control and data

- For control: Polling is "downward", events are "upward"

State feedback loop captures time

- Discrete time, in the case of events
- This is a generalized "state machine"

Each process has exclusive ownership of its data

- The process accepts or rejects transformations

Transformation functions can be pure

- (Though expressing "list of actions" can be hard)

Observations II: connections

- The output of one process is the input of another
- This can be local or distributed
  - reliable or unreliable messages
- Messages are values (copies of data)
  - this is what lets the process reject transformations
- Can limit access to inputs with channel-like mechanisms
  - not "all or nothing" like OO-objects or actors
- Sending/receiving is destructive

Observations III: conclusions

1: The process must be stateful to be useful

2: Messages must carry transformations of state to be useful

Statefulness is usefulness

1+2: Losing/reordering messages is losing/scrambling state

There is no general abstraction for distribution

2: Hierarchies of time

Flavors of unreliability in messages

- Loss
  - Time doesn't happen
- Duplicates
  - Time happens twice
- Reordering
  - Time happens in the wrong sequence

Common: Progression of time is not constant

Using the message-process duality

Solutions to non-constant time:

- Discard non-constant rate messages (lockstep)
  - This is the impossible general abstraction
- Discard non-monotonic messages (clocks or counters)
  - Often viable: eg only need the "latest update"
- Accept a select few non-monotonic messages
  - Possible: Carefully controlled reverse-time (eg "resets")
- Accept all messages (and all inconsistencies)
  - Usually not viable: wrong-order inputs can create wrong-order outputs

The tradeoff in the solutions

- Consistency at the cost of protocol
  - (also at the cost of availability and partition tolerance)
- This protocol is code that must run in the process
  - Responsibility possibly shared among both sender and receiver
- Protocol is limitation
  - Limits the allowable process outputs and state transformations
  - (The "state machine" gets more states)

3: Protocols

What consistency means

In distributed systems

"Replicas agree"

In formal logic

"Free from contradiction"

These are the same thing

1: Allowing some inconsistencies

- Some user-observable inconsistencies might be ok
  - Two elevators arrive vs. no elevator arrives
  - Light turns on but no elevator arrives vs. vice versa
- Hard to reason about
  - Need to consider all possible loss/dup/reorder permutations
  - Need to include node loss too...
    - *In case of combinatorial explosion, look directly at explosion*

2: Collapse loss/dup/reorder space

Want: methodical approach to reduce no. of permutations

Don't care about the order/internals, only the outcomes

Handling:

reordering: receiver is commutative
loss: sender retries often enough
duplicates: receiver is idempotent

Collapsing all permutations is done by:

Only permuting monotonically ("upward counting")

Removing "resets"/"reverse"/negation generally

Removing this has to be done in the state-feedback loop

This is self-referential negation

Enforcing consistency is removing the ability to utter paradoxes

2: Collapsing (cont.)

- Can solve many (most?) problems:
  - Upward-only counters / latest update
  - Set/clear order no. N, then N+1
  - Systems that can work with "eventually consistent" data
- Sometimes too restrictive

3: The middle road

- So far, two extremes:
  - Consider all permutations, or collapse into one
- Alternative:
  - Consider *a few* - how?
- Usually only need very targeted reset/reverse
  - "Reset to initial state", "Go back to 'idle'"
  - Sometimes caused by external factors
    - A node that crashes is "forcibly reset"

Conclusion

Commutativity and idempotency
are the tools for collapsing the permutation space

Complete permutation collapse is only possible
if the nodes cannot fail

The amount of protocol you need
is specification/application dependent


Distributed systems