Two-Way Protocols for occam-π

Adam T. Sampson

Computing Laboratory, University of Kent

University of Kent | Computing

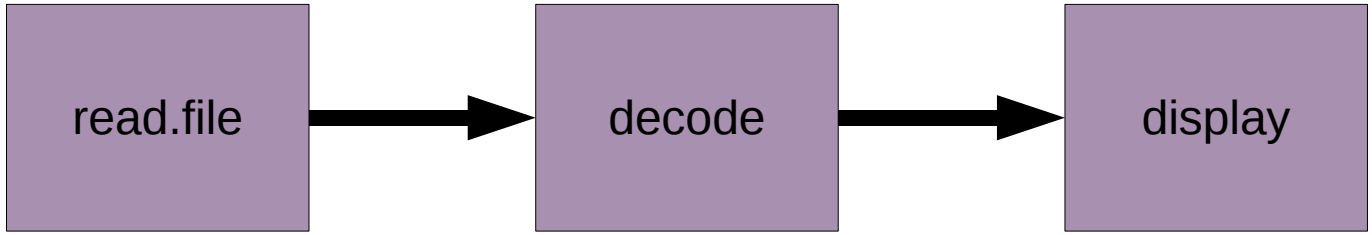
Before we start...

- This is a proposal
  - It hasn't yet been implemented
- It's a synthesis of several existing ideas
- It's applicable to a variety of process-oriented languages and libraries
  - so when I say “occam”, read “occam or JCSP or CHP or PyCSP or ...”

The problem

Processes and channels

- In occam, we build programs by composing *processes* connected by synchronous, unidirectional *channels*



Protocols

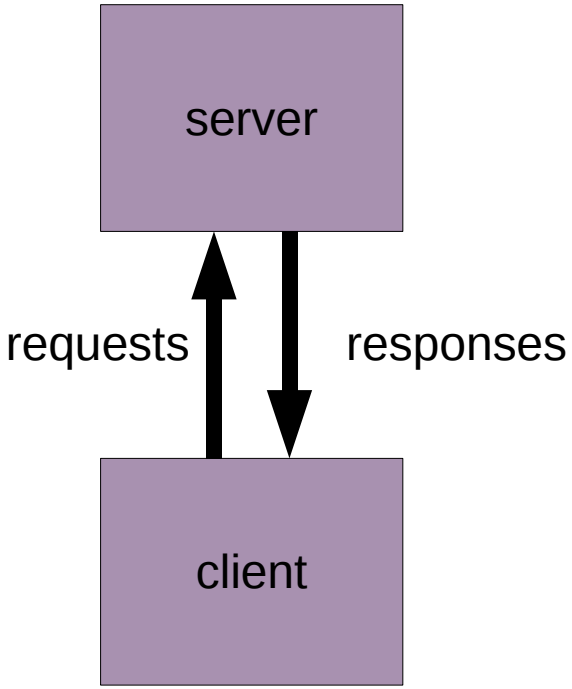
- The messages that may be sent over a channel are defined by a *protocol*
- The compiler checks that the program follows the protocol

```
PROTOCOL POSITION IS INT; INT:

PROTOCOL VIDEO.STREAM
  CASE
    frame; TIME; [][]PIXEL
    end.of.stream
:
```

Clients and servers

- A common design pattern: *server* processes answer requests from *client* processes
- Design rules can be used to construct complex client-server networks safely



Conversations

- Each interaction between a client and server is a *conversation*, and may contain any number of messages
- For example, the *loan pattern*:
  - Client: “Let me borrow your big data structure.”
  - Server: “OK, here it is.”
  - Client: “Right, I'm done; you can have it back now.”

Client-server in occam

- *Request* and *response* channels have separate protocols

```
PROTOCOL LOAN.REQ
  CASE
    borrow
    return; MOBILE DATA
:

PROTOCOL LOAN.RESP
  CASE
    lend; MOBILE DATA
:
```

Safety assured?

- We can check the protocol on each individual channel
- But:
  - Client: “Let me borrow your big data structure.”
  - Server: “OK, here it is.”
  - (Client gets distracted and wanders off.)
  - Client: “Let me borrow your big data structure.”
  - (*Boom!*)

What went wrong?

- Each channel's protocol is checked, but the overall conversation is *not* checked
  - ... so it's possible for the client and server to get into an inconsistent state
- We need a way of describing the *two-way protocol* that the client and server follow
  - This is useful for documentation too!

Some existing approaches

Honeysuckle (Ian East)

- Language for engineering client-server systems
- A *compound service* defines the interface to a server using simplified code

```
sequence
  receive command
  if command
    write
      acquire String
    read
      transfer String
```

Session types (Kohei Honda)

- A formal way of describing two-way communication protocols in terms of the communications that may occur

```
INT! . INT!

(write! . STRING!) | (read! . STRING?)

borrow! . lend? . DATA? . return! . DATA!
```

Session types (Honda)

- Originally proposed for use with the pi-calculus
- Several implementations in various languages
  - For concurrency
  - For network protocols

State machines

- Session types can be *statically* checked by translation into finite state machines
- Session type is a (state machine, state ID) pair
- Communications update the state ID

Proposal

Two-way channels

- Add *two-way channels* to occam-pi
- Can support communication in either direction
  - ... provided both ends agree on the direction
    - You can't ALT between c! and c?
  - Existing channel implementations (CCSP, JCSP et al.) already support this
- Superset of existing channel facilities

Two-way protocols

- Message content and direction is specified using *two-way protocols*
  - These are session type declarations
- Conversations must always be started by the same end...
  - ... so we can *always* tell what direction the next communication will be in
  - This is already one of the client-server design rules: the client must initiate conversation

Splitting up

- In classical occam, one input/output operation performs the whole one-way protocol

```
CHAN POSITION c:
c ! 42; 13
```

POSITION protocol

Splitting up

- Now, a two-way protocol may describe several operations

```
CHAN LEND c:
MOBILE DATA thing:

SEQ
```

```
c ! borrow
c ? lend; thing

-- do something with thing

c ! return; thing
```

LEND protocol

Checking the protocol

- The occam compiler can check this by attaching a session type to each channel end
  - ... which is updated on each communication

```
-- c has session type:
--   lend? . DATA? . return! . DATA!

c ? lend; thing

-- c has session type:
--   return! . DATA!
```

Delegation's what you need

- Since the compiler tracks the session type of each channel end, you can manipulate them safely in the middle of a conversation
  - Abbreviate them
  - Pass them to a procedure
  - For *mobile channel ends*, communicate them to another process
- Can also split a one-way communication across multiple lines

Multiple uses

- Can use this to build client-server systems (as in Honeysuckle)
- But it's not tied to the client-server design rules, so it's useful for other types of process network too
- This can replace several existing uses of *channel bundles* – reduces overhead a bit!

Syntax

Session types in occam

- You'll notice I haven't shown how you define a two-way protocol in occam yet
- There are several possible syntaxes we could consider
- I want to get this *right* – suggestions appreciated!

One approach

- Adapt session types notation into occam syntax
  - This is what most session types implementations do
  - Similar to existing one-way protocol syntax

```
PROTOCOL LOAN IS borrow!;
                 lend?; MOBILE DATA?;
                 return!; MOBILE DATA!:

PROTOCOL STORE IS (read!; STRING?)
                  OR (write!; STRING!):
```

Another way

- Use simplified occam code
  - ... like Honeysuckle does
  - More verbose, but clearer for complex protocols

```
PROTOCOL LOAN
  SEQ
    ! borrow
    ? lend; MOBILE DATA
    ! return; MOBILE DATA
:
```

The problems

- Both approaches have strengths and weaknesses...
  - Describe the lifetime of the channel, or just a single transaction?
  - Reusing and extending protocols
  - Describing a particular state: `LOAN[lend]`
  - Elegance and similarity to existing syntax
- See the paper for more details

Thanks!

- Any questions?