Interfacing C and occam-π

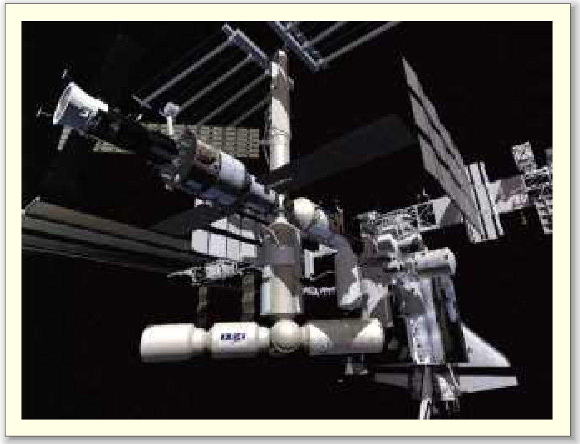
and an “application link layer”

Fred Barnes, Computing Laboratory
University of Kent, Canterbury, UK
F.R.M.Barnes@kent.ac.uk

UNIVERSITY OF KENT

Why C Anyway ?

occam-π


(nasa)

are's CSP
-calculus

hazard
rror

able

(50ns context-switch)

C

evolved from assembler

free-range aliasing through
pointers

structural, imperative

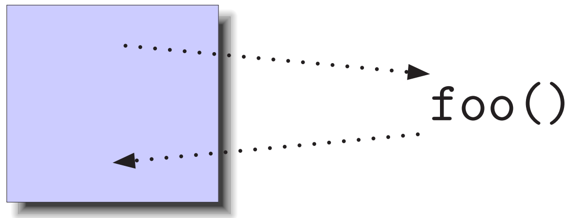
little support for concurrenc


(mission hill)

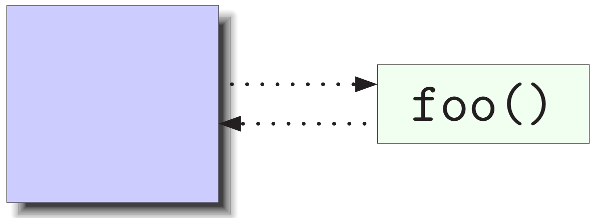
- However, C is widely used and has evolved to fit its usage – principally operating-systems, applications and libraries
- Most language systems provide mechanisms for interfacing with C, typically in a single function-call/return style – occam-π is no exception

Existing occam-π/C Interfaces

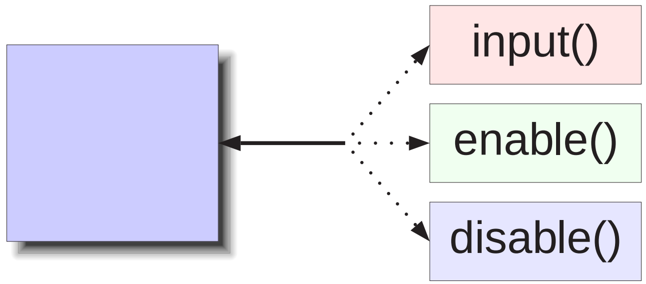
- Plain external C function-calls [Wood, 1998]
  - occam-π processes suspended during the call



- Blocking external calls [Barnes, 2000]
  - call executed in a separate OS thread
  - some overhead in call dispatch/collect/kill



- User defined channels [Barnes, 2002]
  - calls C functions on input/output/ALT
  - mixed blocking/non-blocking



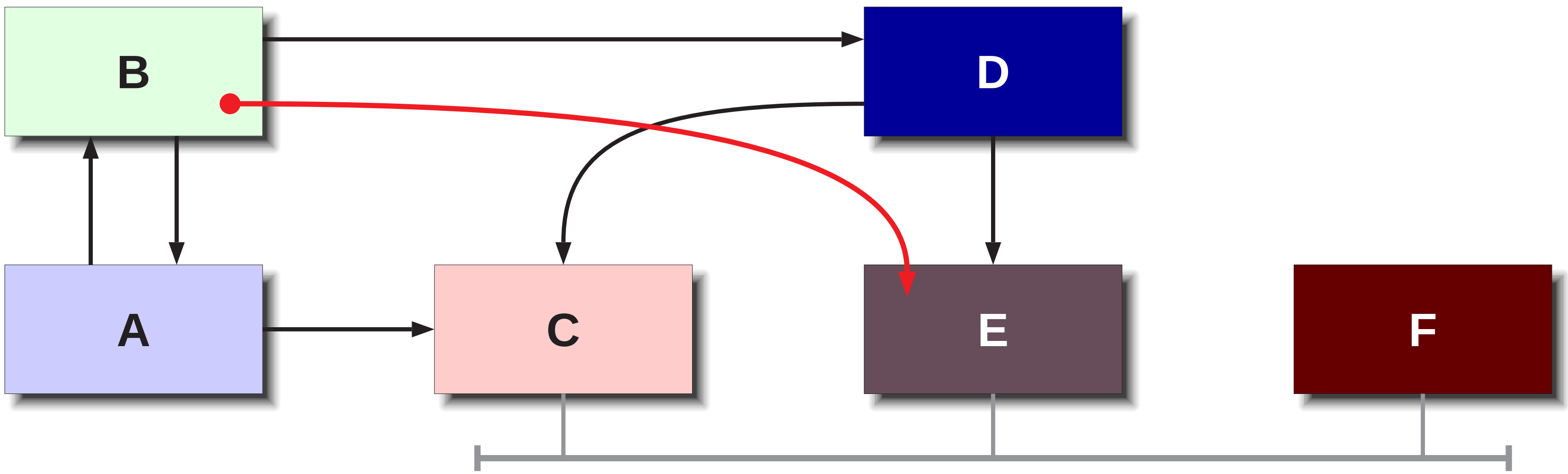
- Main drawback is that storing state between C function calls is awkward — imperative vs. process-oriented incompatibilities

CSP Concurrency in C

- CCSP [Moores, 1999]
  - earlier version of the scheduler used by KRoC for occam-π
  - API similar to Inmos's (for C on the Transputer)
- MESH [Boosten et al., 1999]
  - same code base as CCSP
  - included low-level packet drivers for ethernet chipsets, modelled as channel-communication
- Both good for programming concurrent systems in C, and fairly portable (limited use of in-line assembler)
- But lack the good properties of occam-π— e.g. freedom from parallel aliasing and race-hazard errors (although with good design, scope for these errors is more limited compared with a threads+locks model)

The occam-π C Interface

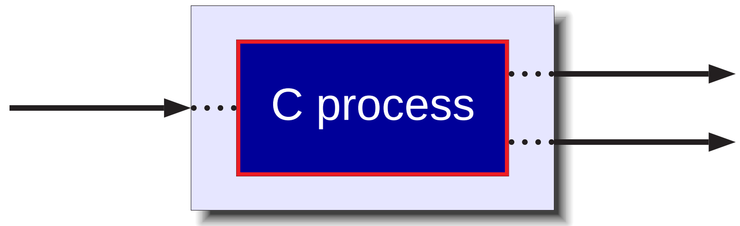
- Provides a mechanism that allows occam-π and C processes to co-exist



- Communicating through ordinary occam-π channels
  - mobile channel-types
  - and synchronising on (mobile) barriers
- Also process priority, extended synchronisations, ...

The occam-π C Interface

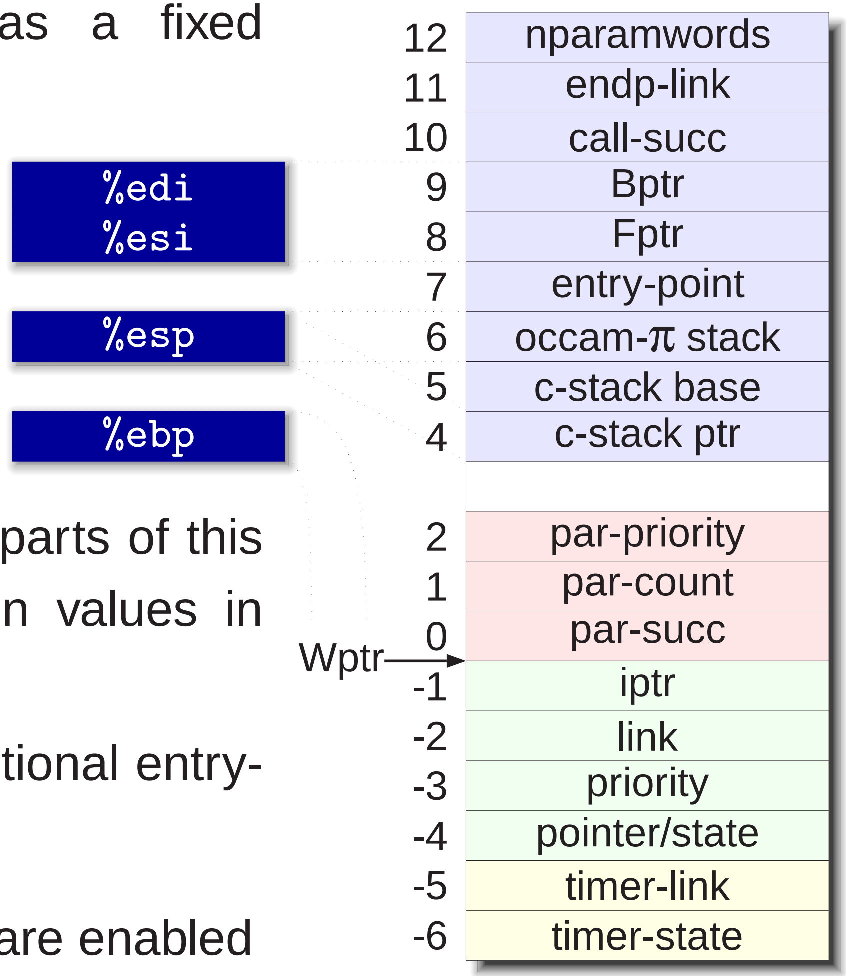
- Works by encapsulating C code inside an occam-π process
  - no changes needed to the run-time kernel
  - slight overhead switching between the C and occam-π contexts
    — comparable to the cost of an ordinary context-switch (sub 100ns)



- All facilities available to occam-π processes are available to their C counterparts
  - provided the appropriate interface code is written — typically in C or assembler
  - approximately 80% of the occam-π functionality is catered for

The occam-π C Interface

- The encapsulating occam-π process has a fixed workspace:
  - standard scheduler state
  - parallel sub-processes state
  - C process state
- C stack size explicitly given
- Overheads incurred when saving/restoring parts of this state — occam-π run-time expects certain values in hardware registers
  - potential for optimisation, by having additional entry-points in the run-time kernel
- Additional overheads when POSIX threads are enabled — used for blocking system-calls



The Application Programming Interface

- Based on the Inmos/CCSP C APIs, extended for occam-π

in? → integrate → out!

```
void integrate (Process *me, Channel *in,
                                  Channel *out)
{
    int total = 0;
    for (;;) {
        int x;
        ChanInInt (in, &x);
        total += x;
        ChanOutInt (out, total);
    }
}
```

integrate

- Invoking the process from occam-π is mildly complicated, really want:

```
#PRAGMA EXTERNAL "PROC CIF.integrate (CHAN INT in?, out!) = 20"
CIF.integrate (c?, d!)
```

(and to automate this too...)

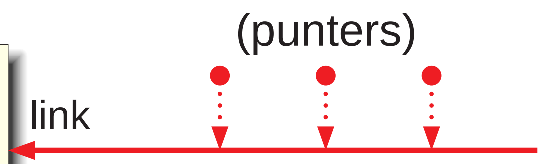
The Application Programming Interface

- Mobile channel-types are mirrored in C structures:

```
typedef struct ct_BAR {
    int refcount;
    Channel serve;
    Channel beer;
    CTSem clisem;
    CTSem svrsem;
} BAR;
```

- C structure reflects the occam-π run-time implementation
- 'refcount' tracks liveness – when it drops to zero, the channel structure should be freed
- 'clisem' and 'svrsem' are the semaphores associated with shared ends

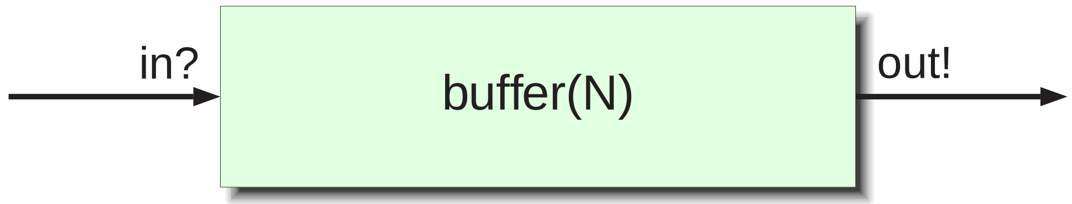
```
void bartender (BAR *link) {
    int job;
    CTSemClaim (&link->svrsem);
    ChanInInt (&link->serve, &job);
    /* process and reply */
    CTSemRelease (&link->svrsem);
}
```




```
link->refcount--;
if (!link->refcount) {
    DMemFree (link);
}
```

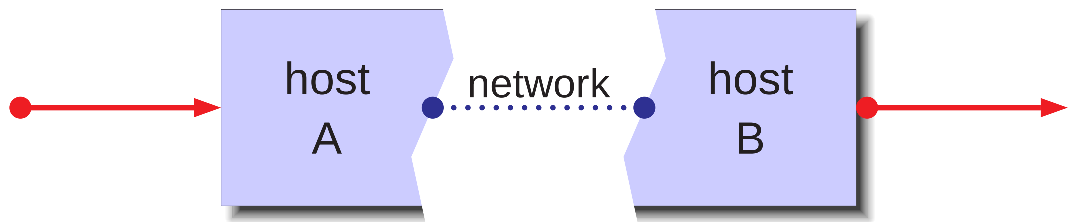
More Interesting Applications

- Because of free-range types in C, can write “generic” components:



- But to do it properly, C needs to be aware of the protocols involved
- This information can currently be generated for mobile channel-types, allowing us to build:

```
CHAN TYPE FOO
  MOBILE RECORD
    CHAN APP.PROTO c?:
:
```

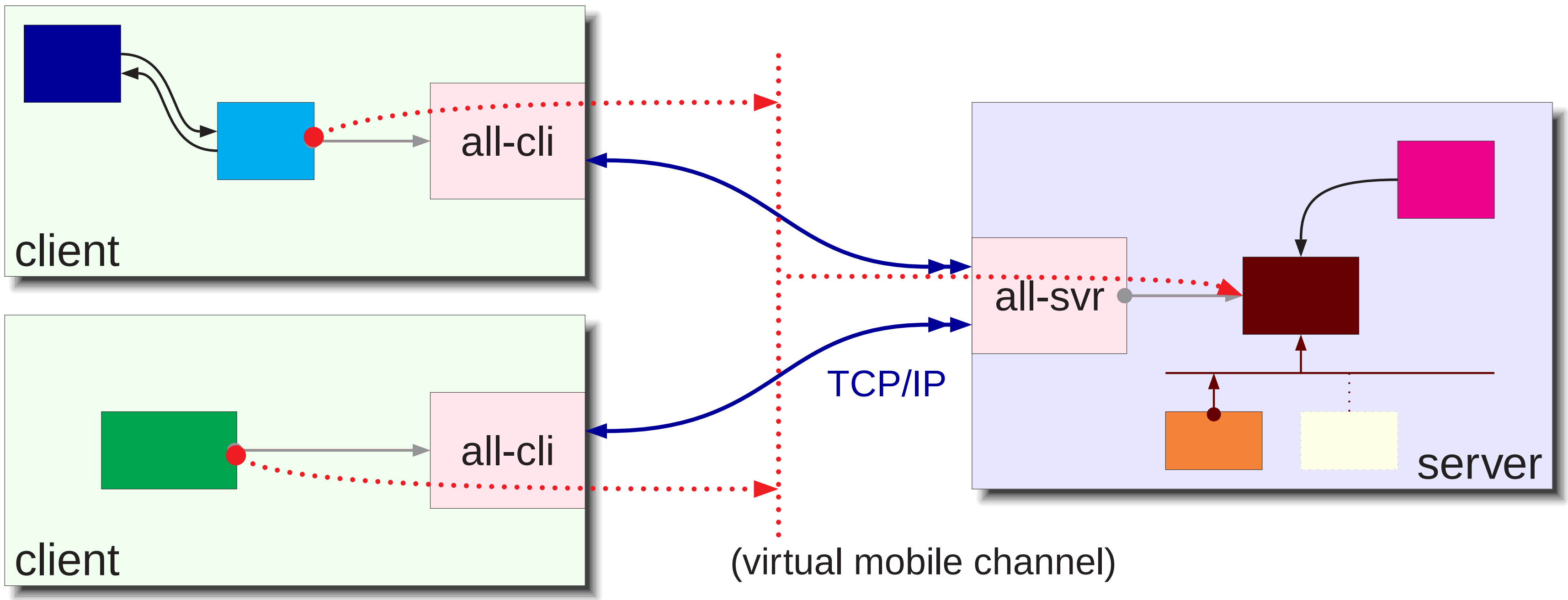


(that can be used with any channel-type)

... networked ?

The Application Link Layer

- A simple (ish) infrastructure for linking mobile channel-types (shared and unshared) over TCP/IP networks
  - in essence a simple version of the "pony" (previously KRoC.net) infrastructure [Schweigler et al., 2003]
- Only supports data communication and single-server arrangements



The Application Link Layer

- The processes 'all.cli' and 'all.svr' are dynamically generated process networks written in C, accessed from occam-π with:

```
PROC all.svr (SHARED MOBILE.CHAN! client, VAL []BYTE sname,
                                usage, ALL.LINK? ctrl, RESULT INT res)
PROC all.cli (MOBILE.CHAN? server, VAL []BYTE sname,
                                         ALL.LINK? ctrl, RESULT INT res)
```

- Compiler generates static type-descriptions of mobile channel-types and the protocols within; C code uses this to setup the infrastructure
- Extra 'usage' parameter to the server defines behaviour:

```
CHAN TYPE S.INFO
  MOBILE RECORD
    CHAN MOBILE []BYTE req?:
    CHAN MOBILE []BYTE resp!:
:
```

```
all.svr (cli, "**:2345",
         "**(0 -> 1)",
              ctrl.svr, res)
```

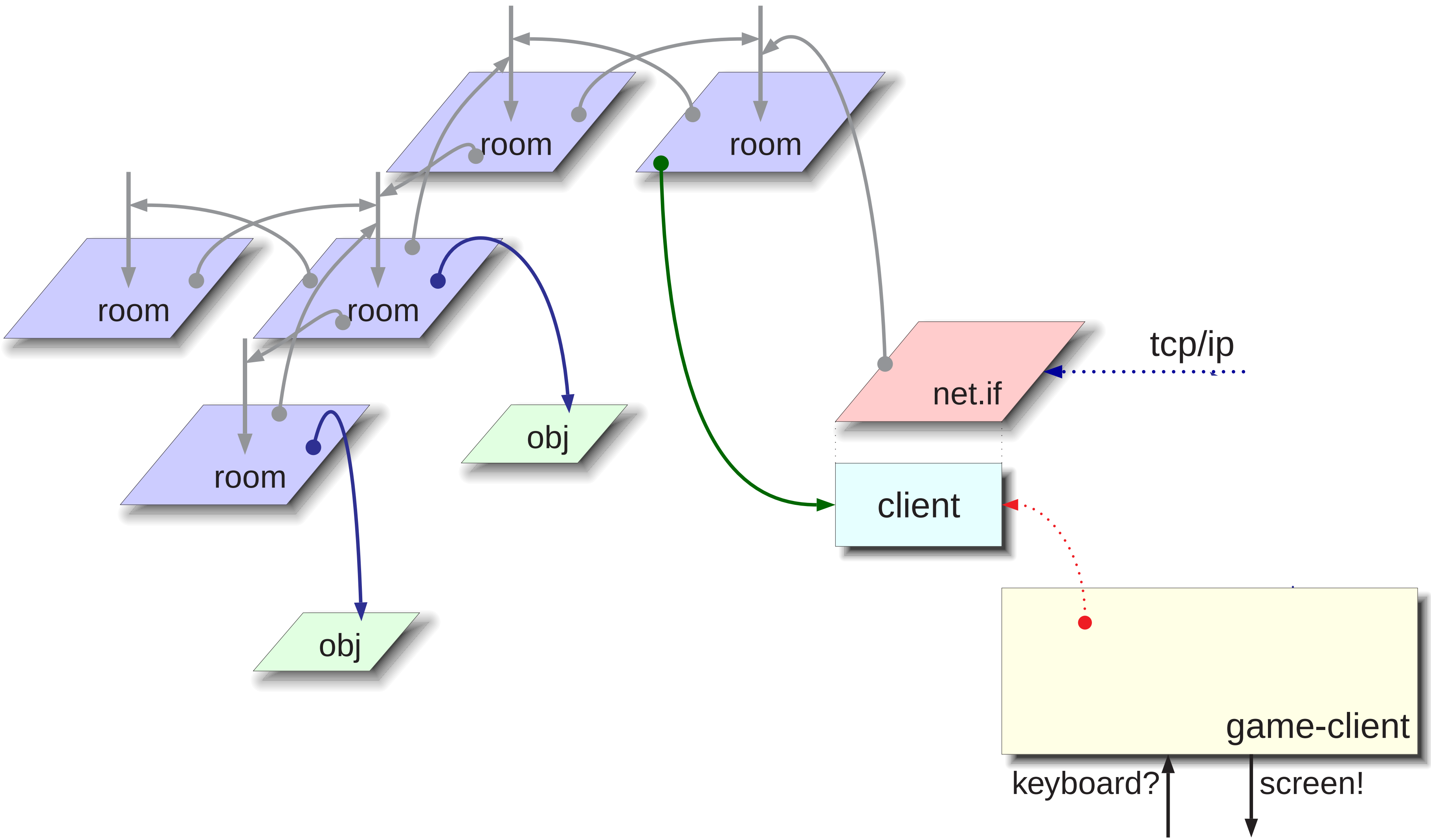
(zero or more, 'req' followed by 'resp')

The Application Link Layer

- Extra control channel-type currently delivers messages
  - future uses include setting keys for encryption, notifies/acceptance of client connections
- Usage information sent when a client connects, used to determine how claims are handled
  - also used to ensure that client and server operation conforms to the specified behaviour — run-time error if they deviate
- Current implementation causes the network to act as a buffer, which would not affect the behaviour of the application in the case of ‘`S.INFO`’
  - clients can send requests at any time
  - server buffers requests, responses sent to the corresponding clients
- Given the amount of socket I/O and pointer manipulation in the link-layer, programming in C is preferable

Conclusions and Further Work

- CIF supports most of the occam-π mechanisms
  - overheads are largely acceptable — approximately 26ns on a 3.2 GHz P4
  - expected to decrease as CIF matures, but never as lightweight (memory)
- Calling between occam-π and C, e.g. for invoking routines in the other language, needs work
  - should be possible to automate given templates of the routines involved
- Application link-layer works `:-)`, but needs more work before distribution
  - first major use will be to better support networked interaction with the “occam adventure” game (from client to game)
  - more generally, provides a quick way of statically distributing occam-π programs over networks
  - more interesting things once mobile processes can be communicated..


room
room
room
room
room
obj
obj
net.if
tcp/ip
client
game-client
keyboard?
screen!