KBUS
A simple messaging system

Tony Ibbs

tibs@tonyibbs.co.uk

http://code.google.com/p/kbus/

KBUS

- A Linux kernel module
- File based:
  - `/dev/kbus0`, etc.
  - `open`, `close`, `read`, `write`, `ioctl`
- Use it:
  - directly
  - via C library
  - via Python API
- Tested using the Python API

With thanks to

Kynesim

Technology development done right.

Two parts

- Simple use of KBUS
- Why KBUS

Simple use:
Senders and Listeners

We start with the simplest form of messaging, with senders and listeners

Terminal 1: Rosencrantz

```
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> from kbus import Ksock
>>> rosencrantz = Ksock(0)
>>> print rosencrantz
Ksock device 0, id 113, mode read/write
```


We start with a single sender, our first “actor”.
The argument to Ksock is the KBUS device number. If KBUS installed, 0 always exists, so is a safe choice.

Terminal 1: Rosencrantz

```
>>> from kbus import Message
>>> ahem = Message('$.Actor.Speak', 'Ahem')
>>> rosencrantz.send_msg(ahem)
MessageId(0, 337)
```


First, create a new Message, with name ‘$.Actor.Speak’ and data ‘Ahem’.
Message names start ‘$.’ and continue with dot-separated parts. Data doesn’t need to be a string!
Sending the message returns its message id as assigned by KBUS.
But no-one is listening...

Terminal 2: Audience

```
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> from kbus import *
>>> audience = Ksock(0)
>>> audience.bind('$.Actor.Speak')
```


So let's add our first listener, an audience, on the same KBUS device – different devices don't communicate.

NB: "`from kbus import *`" is icky. But safe.

They bind to hear all messages called '$.Actor.Speak', whoever sent them.

Terminal 1: Rosencrantz

```
>>> rosencrantz.send_msg(ahem)
MessageId(0, 338)
```

Terminal 2: Audience

```
>>> audience.read_next_msg()
Message('$.Actor.Speak', data='Ahem', from_=113L,
id=MessageId(0, 338))
```


So try again.
This time the audience receives the message.
Note the message id matches.

Terminal 2: Audience

```
>>> print _
<Announcement '$.Actor.Speak', id=[0:338], from=113,
data='Ahem'>
```

Terminal 1: Rosencrantz

```
>>> rosencrantz.ksock_id()
113L
```

Wednesday, 14 July 2010

“print” gives a prettier representation.

A “plain” Message is an Announcement.

The “from” field gives the Ksock id of the sender.

Terminal 2: Audience

```
>>> print audience.read_next_msg()
None
```


If audience tries to listen again, there is nothing else to hear.

Terminal 2: Audience

```
>>> import select
>>> while 1:
...     (r,w,x) = select.select([audience], [], [])
...     # At this point, r should contain audience
...     print audience.read_next_msg()
...
```


We can use “`select`” in the traditional manner (perhaps with more error checking, and even a timeout).

Terminal 1: Rosencrantz

```
>>> rosencrantz.send_msg(Message('$.Actor.Speak',
... 'Hello there'))
MessageId(0, 339)
>>> rosencrantz.send_msg(Message('$.Actor.Speak',
... 'Can you hear me?'))
MessageId(0, 340)
```

Terminal 2: Audience

```
<Announcement '$.Actor.Speak', id=[0:339], from=113,
data='Hello there'>
<Announcement '$.Actor.Speak', id=[0:340], from=113,
data='Can you hear me?'>
```


So now, if Rosencrantz speaks...
...the audience will hear him

Terminal 3: Guildenstern

```
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> from kbus import *
>>> guildenstern = Ksock(0)
>>> print guildenstern
Ksock device 0, id 115, mode read/write
```


Let’s introduce another participant, again on the same KBUS device.

Terminal 3: Guildenstern

```
>>> guildenstern.bind('$.Actor.*')
```

Terminal 2: Audience

```
<CTRL-C>
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>> audience.bind('$.Actor.*')
>>> while 1:
...     print audience.wait_for_msg()
...
```


We can start them listening as well – this time using a wildcard
In retrospect, this makes sense for the audience as well, so let’s fix it.
This time we use the “wait_for_msg” convenience method, instead of “select” directly.

Terminal 1: Rosencrantz

```
>>> rosencrantz.bind('$.Actor.*')
```


It seems likely that Rosencrantz will want to hear his colleague, so let's do the same

Terminal 3: Guildenstern

```
>>> guildenstern.send_msg(Message('$.Actor.Speak',
'Pssst!'))
MessageId(0, 341)
>>> print guildenstern.read_next_msg()
<Announcement '$.Actor.Speak', id=[0:341], from=115,
data='Pssst!'>
```


So let Guildenstern speak
Note he hears himself, since he is listening to '$.Actor.*' messages

Terminal 1: Rosencrantz

```
>>> msg = rosencrantz.read_next_msg()
>>> print msg
<Announcement '$.Actor.Speak', id=[0:341], from=115,
data='Pssst!'>
```

Terminal 2: Audience

```
<Announcement '$.Actor.Speak', id=[0:341], from=115,
data='Pssst!'>
<Announcement '$.Actor.Speak', id=[0:341], from=115,
data='Pssst!'>
```


And Rosencrantz hears
And so does the audience – twice
...once for each binding: ‘$.Actor.Speak’ and ‘$.Actor.*’

Terminal 2: Audience

```
<CTRL-C>
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File ".../ksock.py", line 492, in wait_for_msg
    (r, w, x) = select.select([self], [], [], timeout)
KeyboardInterrupt
>>> audience.unbind('$.Actor.Speak')
>>> while 1:
...     msg = audience.wait_for_msg()
...     print msg
...
```


The solution is simple - unbind one of the bindings.
Note the unbinding must match exactly.

Simple use:
Requests and Repliers

Wednesday, 14 July 2010

We’ve shown how to “announce” (or perhaps “shout”) messages, but KBUS also supports asking questions.

Terminal 3: Guildenstern

```
>>> guildenstern.bind('$.Actor.Ask.Guildenstern', True)
```


Guildenstern binds as a Replier on ‘$.Actor.Ask.Guildenstern’.
Only one person may be bound as a Replier for a given message name at any one time, so that it is unambiguous who is meant to reply.

Terminal 1: Rosencrantz

```
>>> from kbus import Request
>>> req = Request('$.Actor.Ask.Guildenstern',
... 'Were you speaking to me?')
>>> rosencrantz.send_msg(req)
MessageId(0, 342)
```


So Rosencrantz can build a Request, and send it.
It is an error to send a Request when there is no Replier bound to that message name (contrast with ordinary messages)

Terminal 1: Rosencrantz

```
>>> print rosencrantz.read_next_msg()
<Request '$.Actor.Ask.Guildenstern', id=[0:342],
from=113, flags=0x1 (REQ), data='Were you speaking to
me?'>
>>> rosencrantz.unbind('$.Actor.*')
```


Of course, Rosencrantz still gets a copy, as he bound to the wildcarded ‘$.Actor.*’.
We can stop that by unbinding from the wildcard.

Terminal 3: Guildenstern

```
>>> req = guildenstern.read_next_msg()
>>> print req
<Request '$.Actor.Ask.Guildenstern', id=[0:342],
from=113, flags=0x3 (REQ,YOU), data='Were you speaking
to me?'>
>>> print req.wants_us_to_reply()
True
```


Guildenstern receives the request.
It is marked REQ to show it is a Request, and YOU to show this recipient should reply.
There is a convenience method for the latter.

Terminal 3:Guildenstern

```
>>> msg = guildenstern.read_next_msg()
>>> print msg
<Request '$.Actor.Ask.Guildenstern', id=[0:342],
from=113, flags=0x1 (REQ), data='Were you speaking to
me?'>
>>> guildenstern.unbind('$.Actor.*')
```


Guildenstern also gets another copy, because of binding to ‘$.Actor.*’
This copy is marked REQ (it is still a Request), but not YOU.
The obvious thing to do is to undo that binding.
NB: The YOU message is always guaranteed to arrive before any other “copies”.
(maybe mention “want_messages_once()” as an alternative approach)

Terminal 3: Guildenstern

```
>>> rep = reply_to(req, 'Yes, yes I was')
>>> print rep
<Reply '$.Actor.Ask.Guildenstern', to=113, in_reply_to=
[0:342], data='Yes, yes I was'>
>>> guildenstern.send_msg(rep)
MessageId(0, 343)
>>> guildenstern.read_next_msg()
```


Regardless, Guildenstern can create a reply,
using the convenience function “reply_to()”, which fills in the appropriate “to” and “in_reply_to” fields, and then send it.
Note we don’t get a copy, as we’re replying, not sending.

Terminal 1: Rosencrantz

```
>>> rep = rosencrantz.read_next_msg()
>>> print rep
<Reply '$.Actor.Ask.Guildenstern', id=[0:343], to=113,
from=115, in_reply_to=[0:342], data='Yes, yes I was'>
```


And Rosencrantz receives the answer.
Note he didn’t need to be bound to the message name to receive a reply.
Also, KBUS goes to some lengths to guarantee he WILL get a reply, even if Guildenstern “went away”.

Terminal 2: Audience

```
<Request '$.Actor.Ask.Guildenstern', id=[0:342],
from=113, flags=0x1 (REQ), data='Were you speaking to
me?'>
<Reply '$.Actor.Ask.Guildenstern', id=[0:343], to=113,
from=115, in_reply_to=[0:342], data='Yes, yes I was'>
```


And the audience sees the dialogue.

Simple use:
Stateful Requests

Sometimes, it is useful to preserve state in the Replier.

Terminal 1: Rosencrantz

```
>>> req = Request('$.Actor.Ask.Guildenstern',
>>> 'Will you count heads for me?')
>>> rosencrantz.send_msg(req)
MessageId(0, 343)
```


So Rosencrantz sends his request.

Terminal 3: Guildenstern

```
>>> req = guildenstern.read_next_msg()
>>> guildenstern.send_msg(reply_to(req, 'I shall'))
MessageId(0, 345)
>>> guildenstern.bind('$.Actor.CoinToss', True)
>>> heads = 0
>>> while True:
...   toss = guildenstern.wait_for_msg()
...   print toss
...   if toss.data == 'Head':
...     print 'A head - amazing'
...     heads += 1
...   else:
...     print 'Bah, tails'
...   rep = reply_to(toss, 'Head count is %d'%heads)
...   guildenstern.send_msg(rep)
...
```


Guildenstern agrees, and prepares to count the number of heads.

Terminal 1: Rosencrantz

```
>>> rep = rosencrantz.read_next_msg()
>>> print rep.from_
115
>>> # Throws a head
... from kbus import stateful_request
>>> sreq = stateful_request(rep, '$.Actor.CoinToss',
... 'Head')
>>> print sreq
<Request '$.Actor.CoinToss', to=115, flags=0x1 (REQ),
data='Head'>
>>> rosencrantz.send_msg(sreq)
MessageId(0, 346)
```


Rosencrantz notes Guildenstern's Ksock id (from his reply).
He creates a stateful request (with the convenient function, based on the Reply) which says who it is TO.
The send will fail if the named sender is no longer the Replier bound to this message name.

Terminal 3: Guildenstern

```
<Request '$.Actor.CoinToss', id=[0:346], to=115,
from=113, flags=0x3 (REQ,YOU), data='Head'>
A head - amazing
MessageId(0, 347)
```

Terminal 1: Rosencrantz

```
>>> count = rosencrantz.read_next_msg()
>>> print 'So,',count.data
So, Head count is 1
>>> # Throws a head
... sreq = stateful_request(rep, '$.Actor.CoinToss',
... 'Head')
>>> rosencrantz.send_msg(sreq)
MessageId(0, 348)
```


And so with another coin toss - another head...
Again, remembering to use a stateful request

Terminal 3: Guildenstern

```
<Request '$.Actor.CoinToss', id=[0:348], to=115,
from=113, flags=0x3 (REQ,YOU), data='Head'>
A head - amazing
MessageId(0, 349)
```


Which Guildenstern faithfully counts

Terminal 1: Rosencrantz

```
>>> count = rosencrantz.read_next_msg()
>>> print 'So,',count.data
So, Head count is 2
>>> # Throws a head
```


And again, Rosencrantz throws a head – he's good at this

Terminal 3: Guildenstern

```
<CTRL-C>
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File ".../ksock.py", line 492, in wait_for_msg
    (r, w, x) = select.select([self], [], [], timeout)
KeyboardInterrupt
>>> print 'Falstaff! No! Ouch!'
Falstaff! No! Ouch!
>>> guildenstern.close()
```


But Falstaff intervenes,
and forces Guildenstern to disconnect from his Ksock.

Terminal 4: Falstaff

```
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> from kbus import *
>>> falstaff = Ksock(0)
>>> falstaff.bind('$.Actor.CoinToss', True)
```


and Falstaff binds as Replier instead.

Terminal 1: Rosencrantz

```
... sreq = stateful_request(rep, '$.Actor.CoinToss',
... 'Head')
>>> rosencrantz.send_msg(sreq)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../ksock.py", line 432, in send_msg
    return self.send()
  File ".../ksock.py", line 220, in send
    fcntl.ioctl(self.fd, Ksock.IOC_SEND, arg);
IOError: [Errno 32] Broken pipe
```


but because he does not have the same Ksock id, the stateful request fails.
The Python interface isn't very helpful with the IOError numbers that KBUS uses...

```
$ errno.py 32
Error 32 (0x20) is EPIPE: Broken pipe

KBUS:
On attempting to send 'to' a specific replier, the replier with
that id is no longer bound to the given message's name.
```


but there is a useful command line utility, giving the standard Unix “errno” value, and the KBUS usage.

Terminal 2: Audience

```
<Request '$.Actor.Ask.Guildenstern', id=[0:344],
from=113, flags=0x1 (REQ), data='Will you count heads
for me?'>
<Reply '$.Actor.Ask.Guildenstern', id=[0:345], to=113,
from=115, in_reply_to=[0:344], data='Yes, yes I shall'>
<Request '$.Actor.CoinToss', id=[0:346], to=115,
from=113, flags=0x1 (REQ), data='Head'>
<Reply '$.Actor.CoinToss', id=[0:347], to=113,
from=115, in_reply_to=[0:346], data='Head count is 1'>
<Request '$.Actor.CoinToss', id=[0:348], to=115,
from=113, flags=0x1 (REQ), data='Head'>
<Reply '$.Actor.CoinToss', id=[0:349], to=113,
from=115, in_reply_to=[0:348], data='Head count is 2'>
```


And, from the audience's point of view

Why KBUS

Why

- We work in the embedded world
- We want a means of communication between software elements
- We've had experience of bad solutions

Set Top Boxes, etc.
Communicate between GUI, remote control, keyboard, video and audio decoders, demuxers, recording software, etc.

Bad things

- Race conditions when either end restarts
- Unreliability
- Poor documentation

Aims

- Simple models to "think with"
- Predictable delivery
- *Always* get a reply to a request
- Deterministic message order on a bus
- Small codebase, in C
- Easy to use from Python (well, I want that)
- Open source

Reliable and well-documented as well.
There didn't seem to be any option but to write our own.

Simple models: naming

- Ksock
- Sender, Listener, Replier
- Message, Announcement, Request, Reply
- "`$.message.name`"

Wednesday, 14 July 2010

Not very happy with Ksock, but better than "Elephant"
I'm sorry for Limpets (which we may get to)

Simple models: APIs

- The “bare Unix” level
- The C library - hides the details
- The Python API - even better

The C library hides the mess of handling errno from ioctls, etc.
The Python API was written as the primary testing API.

Simple models: Data

KBUS does not say anything about the data being transferred

Predictable delivery

- It is acceptable for a Listener to miss messages

  (although they should be able to avoid it)

- But it is not acceptable for a Replier to miss a Request

- And each Request *shall* produce a Reply

A “send” fails if:

- the sender of a Request has a full queue

  `(-ENOLCK)`

- the receiver of a Request has a full queue

  `(-EBUSY)`

- a message is marked “ALL or FAIL” and any of the listeners could not receive it

  `(-EBUSY)`

- a message is marked “ALL or WAIT” and any of the listeners could not receive it

  `(-EAGAIN)`

Wednesday, 14 July 2010

We believe the last two are mainly of use for debugging.
After -EAGAIN, the sender then needs to discard the message, or play the poll/select game to wait for the send to finish.

"""*KBUS guarantees that each Request will (eventually) be matched by a consequent Reply (or Status) message, and only one such.*"""

If the replier can't give a Reply, KBUS will generate one - for instance:

- "`$.KBUS.Replier.Unbound`" or
- "`$.KBUS.Replier.GoneAway`"

From the documentation (sorry about the phrasing)
If KBUS generated it, "from" will be 0

Kernel module

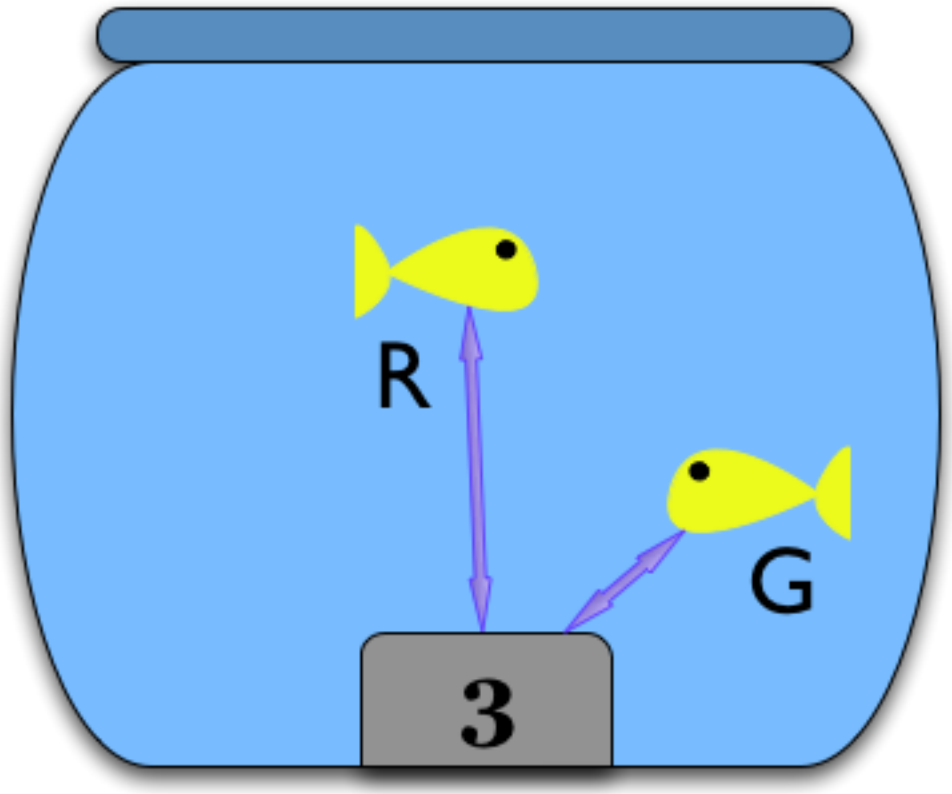
- we can have a reliable file interface
- but the kernel simplifies it for us
- guaranteed to know when a listener goes away (Ksock closes)
- realistic expectation of reliability
- kernel datastructures
- kernel memory handling

The kernel does stuff for us

- Requires predictable interfaces
- Enforces a coding style
- We can “ignore” threading, multiple CPUs, etc
- We should have less context switching
- We can try submitting it

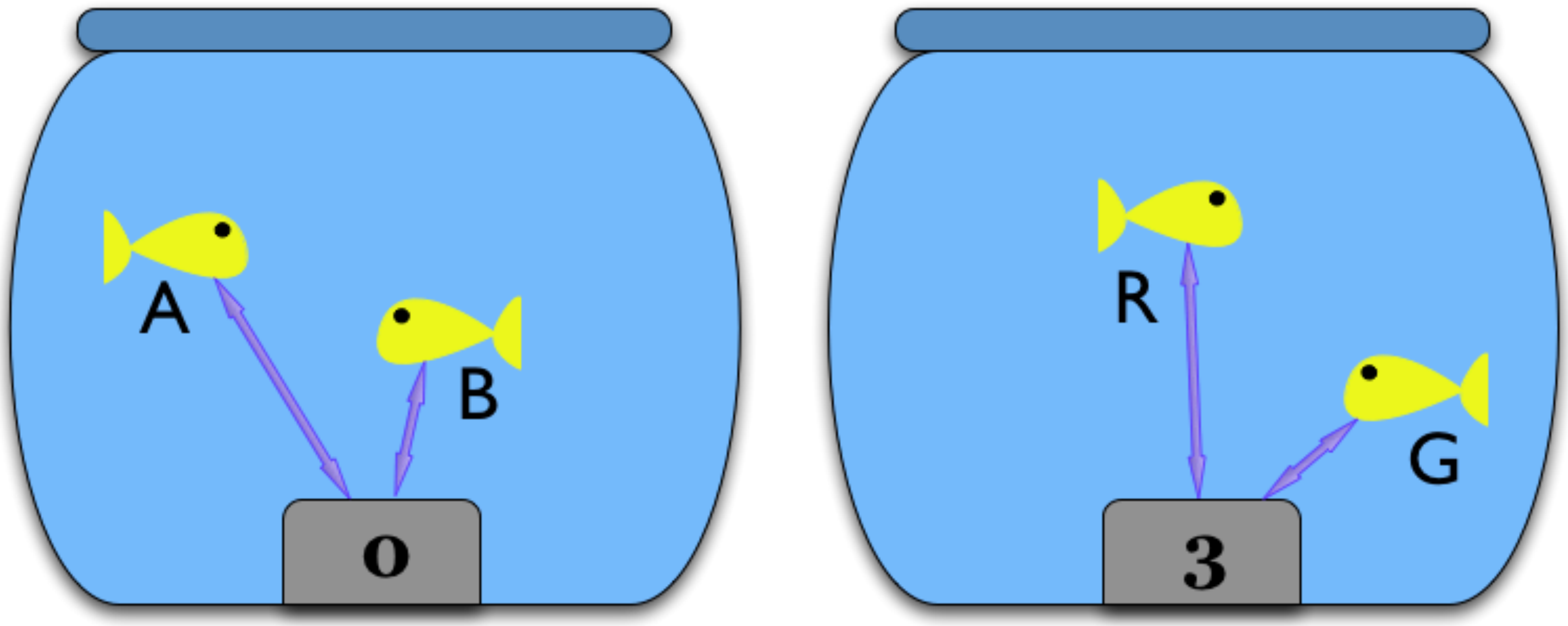
That last is scary but likely to be very valuable – even if it is not accepted, KBUS will probably be improved.

Isolation



Each KBUS is isolated from the others, as if it were in a metaphorical goldfish bowl.

Apologies that this R and G are using KBUS device 3, not the expected 0



Two other fish, communicating via a different KBUS device, are in a different metaphorical bowl, and thus cannot communicate with R and G.

Example uses

- Set Top Box - Instructions from user interface to:
  - receiver (change channel, volume)
  - recorder (play, rewind, pause, record)
  - DRM interface
- Industrial control systems
- Remote control and telemetry

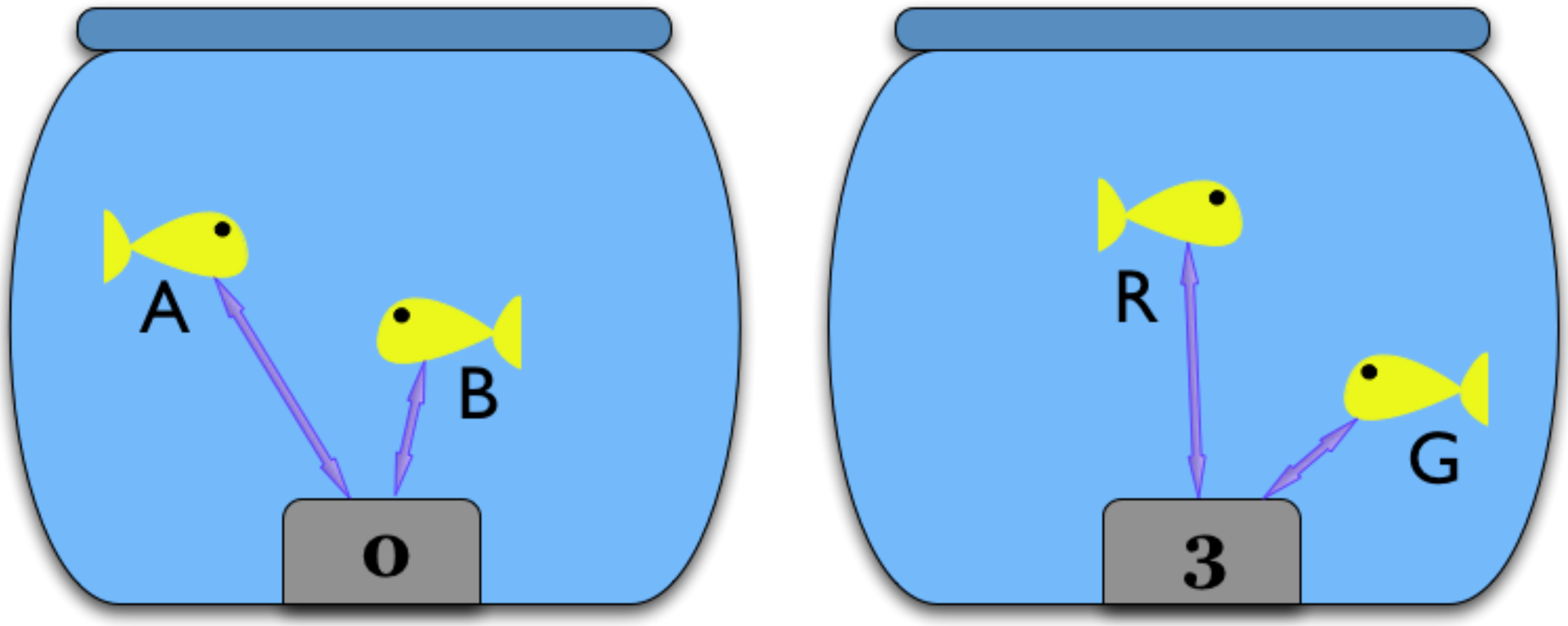
Any alternatives?

- POSIX message queues (`mqueue`)
  (new in 2.6.2, limited resource usage, too simple)
- DBUS
  (complex, no message ordering, large)
- zeromq (`0mq`)
  (pretty, pragmatic, C++, deliberately "simpler")
- what else?

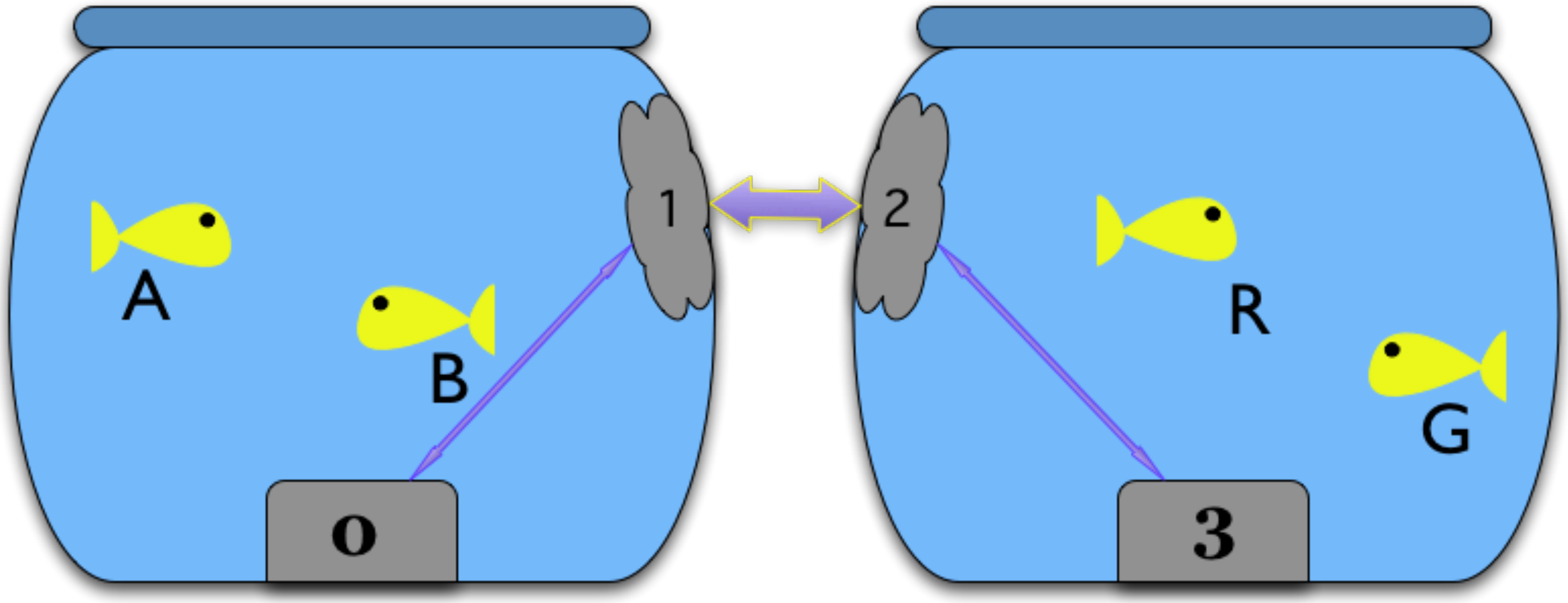
Fin

OK, Limpets...

A Limpet proxies KBUS messages between a Ksock and another Limpet.



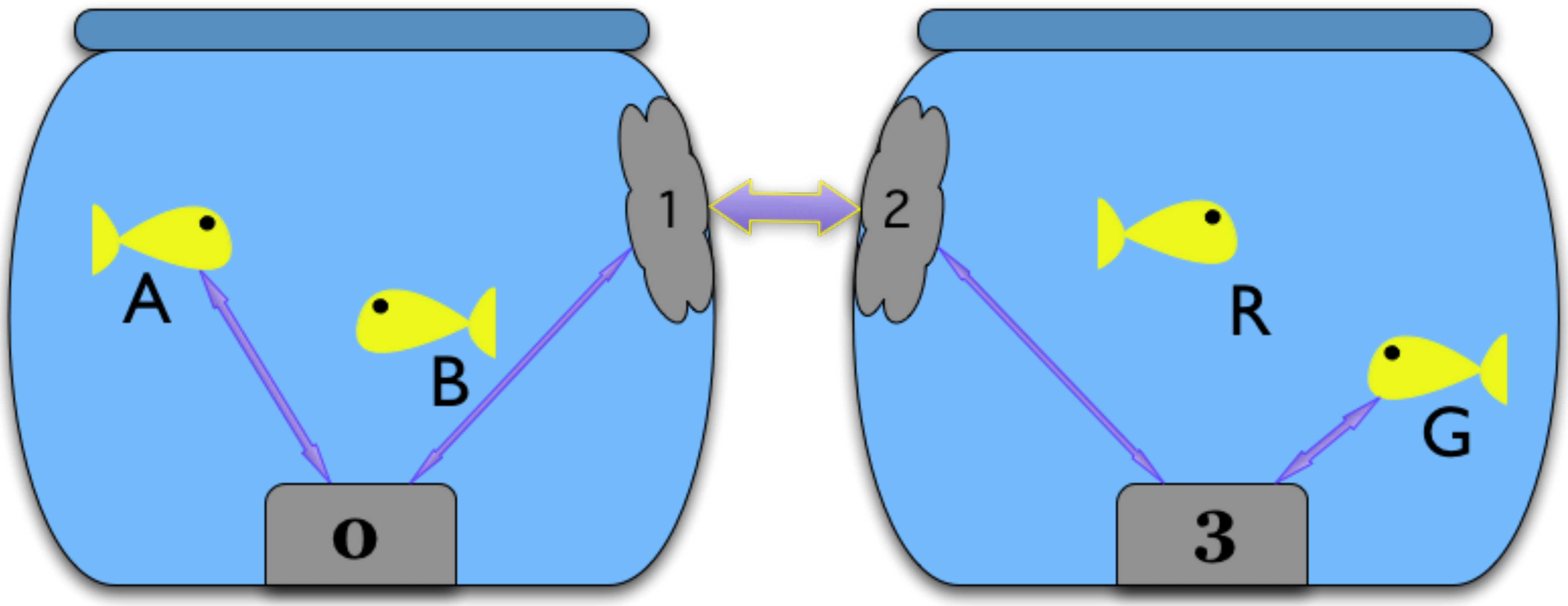
Our isolated bowls



A pair of Limpets

Wednesday, 14 July 2010

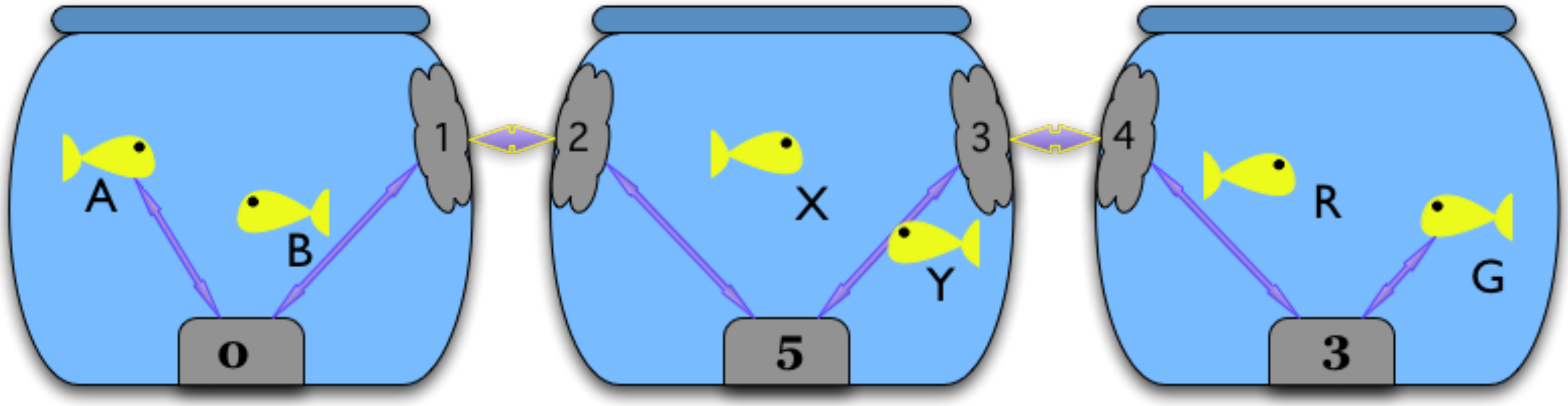
The two limpets talk to each other by some means - perhaps using lasers. Lasers are good. KBUS doesn't mandate how this is done, but does provide the KBUS/Limpet mechanism.



A talks to G

Wednesday, 14 July 2010

Message passing should act as transparently as possible.
A and G think they're in the same bowl
If A binds to hear "$.X", then Limpet 1 tells Limpet 2 to bind for the same, and when G sends "$.X", Limper 2 hears, passes it back to Limpet 1, who "says" it again for A to hear.



Even with intermediate bowls

Really Fin