Machinetalk Tutorial

Michael Haberler

what we can do here

- understand what is broken
- overview of the replacement stack
- discuss key parts
- assemble into examples
- lay groundwork for ‚remote UI‘ talk

why break it - it sortof works, right?

- HAL: great, but zero provisions for remote ops
- the EMC distributed architecture was let rot away
- NML: click here for full list of NML misfeatures
- verdict: beyond repair - NML/RCS stack, EMC_STATUS has to go.
- this is a repair job, once per decade

NML bug list

- serialisation: unsuited for RT usage (C++ objects - transcoding required at RT boundary, so uninterpreted end2end message contents not possible; see also verious warts and workarounds for error messages out of RT)
- arbitrary limits: 'buffer size' (nowadays 16k or so, config time)
- manual serialisation - defining a new message is absurdly complex and error prone
- no way to support optional or repeated fields in a message, and hence detect presence/repeat count on receiving end
- no way to support arbitrary length fields in a message
- no support for NULL values (i.e. not 'zero', but 'non-existent')
- no introspection support *)
- no versioning support/message extensions where old code at least understands the base part
- all manual language bindings - no complete EMC NML bindings available even today for say Python because the effort is absurd
- IDL support vanished in the mists of history

- transport: rather fuzzy and low level semantics - the 'M' in 'NML' stands for 'message' which is usually understood to be a record which is subject to queued transactional interaction, but what the model really is is a named shared memory buffer (except for the error message queue), with all sorts of funny effects if writers dont behave
- no transport abstraction to interact with RT (besides serialisation limitations mentioned above)
- no abstraction to support message-based interaction between RT components
- no support for common interaction patterns like remote procedure call or publish/subscribe **)
- no routing support in presence of multiple consumers/multiple producers
- no clear concept of how to indicate 'message consumed' - see assorted problems with serial_number
- consumption semantics: no way to publish a message to more than one subscriber - first consumer wins, others dont see the message
- network capability: decayed to the state that we dont seem to know if it actually works anymore
- unmaintained and unused outside linuxCNC
- code base readability: no polite comment I could think of
- low level of abstraction - using code repeats long wads of incantations, for instance buffer attach at startup
- polling is the only way to detect progress - no blocking waits, no callbacks, no reactor pattern
- no integration in event loops like gevent, qt, libev, poll(2), eventfd etc, making integration into UI's yet another polling exercise
- abysmal performance due to fixed polling cycles and associated overhead shortening the same if more punch is needed - see for instance the 'emctaskeager' variable in task for a related hackaround
- no support for encryption and authentication
- no support for peer autodiscovery
- nml config, linuxcncserver - not sure what value this adds, other stacks work perfectly fine without any of these

what we leave behind: the one-page intro to RCS and NML

- RCS (transport): named, fixed size shared memory buffers: „command“, „status“, „error“
- ops: atomic read(), write(), peek()
- NML (content encoding): really just C POD structs disguised as C++
- cost: unmaintained 36KLOC C++

machinekit vision
(mah edition):

- a portable, distributed realtime toolkit supporting arbitrary UI's - beyond gtk2+Tk
- a great CNC application on top

- ‚UBC' covered ‚portable realtime'
- machinetalk covers ‚portable distributed'
  + ‚support arbitrary UI technologies'

requirements

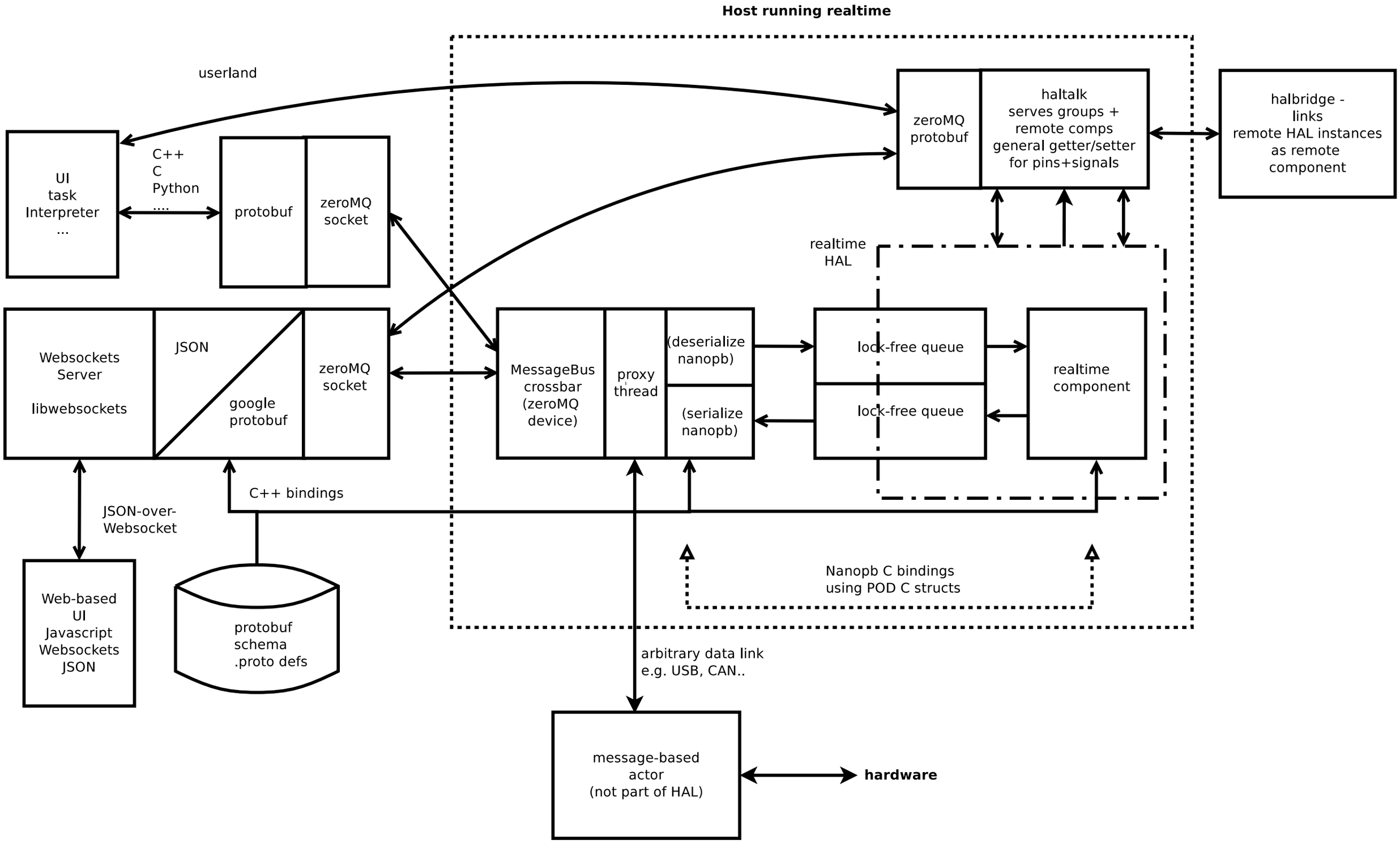
- language support: minimum C,C++, Python
- fitness for RT
- no proxies, „translators“ etc - end2end
- no arbitrary limitations; architecture independent
- blocking/non-blocking messaging support
- support any startup/restart order
- idempotent connect/reconnect
- no brokers (like RabbitMQ, AMPQ..)
- interface description language
- introspection, versioning support
- minimum configuration - autodiscovery
- must be „web-friendly“ (JSON/Websockets)

concern	addressed by
service discovery + identification	zeroconf, URIs
talking to Web UI's	websockets/JSON
message content	protobuf
getting from A to B	zeroMQ
RT scalar I/O	group + rcomp service
RT command/response I/O	*messagebus service*, ringbuffer API
configuration state	*config service*

the machinetalk „service“

- addresses a concern:
  - report/set HAL objects, send motion commands, tap into logging stream..
    - 1..n using entities
- URI, interaction pattern, message scheme
- name - unit of announcement
- *where authentication + credentials happen*

Machinetalk architecture



the set of spares:

- HAL API extensions: ringbuffer, groups, remote components
- message encoding - protobuf
- transport - zeroMQ
- (discovery - zeroconf)

ringbuffers (aka FIFOs/queues)

- shared memory, read/write sides
- nonblocking ops only, multicore-safe
- zero-copy & copying API calls: test data available/space, read, write
- byte stream
- record oriented: preserves boundaries
- multiframe - list of record/flag tuples
- src/rtapi/ring.h - static inlines; no OS support required
- 140-300nS/ringop
- usage: standalone - unnamed

HAL named ringbuffers

- outside components - halcmd support
- user/RT; user/user; RT/RT
- API (C, C/RT, Python):
- attach to a named ring
- create a ring of size, attributes
- inspect type etc
- record_availble(), read()
- space_available(), write()

record ops, reader, userland:

```
import time
from machinekit import hal

name = "ring1"
ringsize = 16384
polltime = 0.1

if name in hal.rings():
    r = hal.Ring(name)
else:
    r = hal.Ring(name, size=ringsize)

while True:
    record = r.read()
    if record is None:
        time.sleep(polltime)
    else:
        print "got: ", record.tobytes()
        r.shift() # consume
```

src/machinetalk/tutorial/ringbuffers/reader-singleframe.py

record ops, writer, userland:

```
import sys
from machinekit import hal

name = "ring1"
ringsize = 16384
polltime = 0.1

try:
    # to attach if no size given
    r = hal.Ring(name)
except RuntimeError,e:
    # else create
    r = hal.Ring(name, size=ringsize)

count = 10
for n in range(count):
    try:
        r.write("record %d" % n)
    except RuntimeError,e:
        print e
```

src/machinetalk/tutorial/ringbuffers/writer-singleframe.py

halcmd ring support:

```
halcmd: show ring
Rings:
Name                Size       Type    Rdr Wrt Ref Flags
ring1               16384      record 0/0 0/0 1   recmax:16368

halcmd: newring foo 4096 stream
halcmd: show ring
Rings:
Name                Size       Type    Rdr Wrt Ref Flags
foo                 4096       stream 0/0 0/0 0   free:4095
ring1               16384      record 0/0 0/0 1   recmax:16368
```

HAL signal groups

- group a set of signals into a named unit of change detection and status reporting
- example: DRO: track current X, Y, Z -> ‚position‘
- scaling - scan once, use many

```
# 1. signals as taken from configs/common/core_stepper.hal:
net Xpos-fb stepgen.0.position-fb => axis.0.motor-pos-fb
net Ypos-fb stepgen.1.position-fb => axis.1.motor-pos-fb
net Zpos-fb stepgen.2.position-fb => axis.2.motor-pos-fb

# 2. declare a HAL group 'position', scan every 100mS
newg position 100
newm position Xpos-fb
newm position Ypos-fb
newm position Zpos-fb

# 3. make haltalk report this group:
loadusr  haltalk --groups position # <... other groups to report>
```

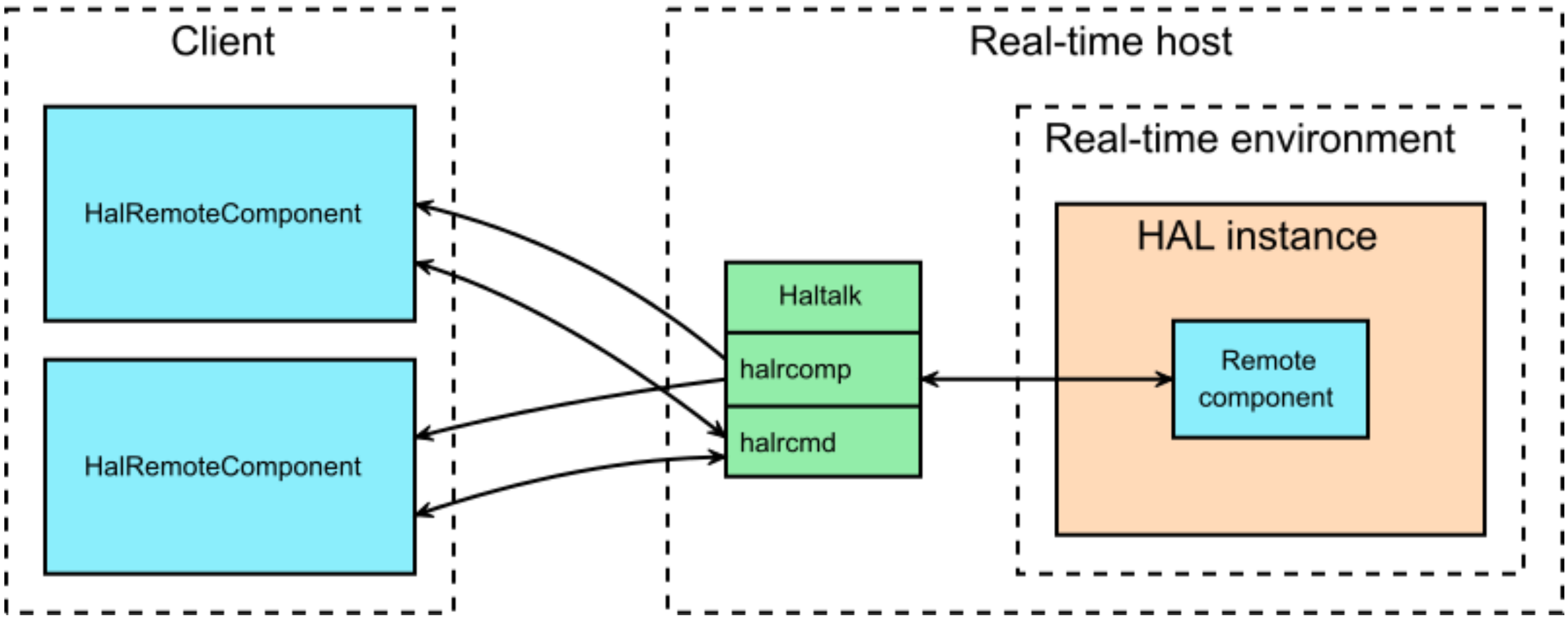
remote HAL components

- component with name, pins but NO local thread function
- „stands in“ for remote entity - like a UI
- decouple pin definition, thread function code (remote) - remember POSTGUI_HALFILE?
- HAL support for declaring and serving
- pin change detection and reporting
- usually served by haltalk

example remote comp

```
sete 1 0.001
newcomp motorctrl timer=100 # acceptdefaults
newpin motorctrl motorctrl.cmd_maxacc    float out
newpin motorctrl motorctrl.cmd_maxvel    float out
newpin motorctrl motorctrl.cmd_pos       float out
newpin motorctrl motorctrl.fb_acc        float in eps=1
newpin motorctrl motorctrl.fb_acc.scale  float in eps=1
newpin motorctrl motorctrl.fb_pos        float in eps=1
newpin motorctrl motorctrl.fb_pos.scale  float in eps=1
newpin motorctrl motorctrl.fb_vel        float in eps=1
newpin motorctrl motorctrl.fb_vel.scale  float in eps=1
newpin motorctrl motorctrl.led1          bit in
newpin motorctrl motorctrl.lowpass_gain  float out
newpin motorctrl motorctrl.scope_trigger bit out
newpin motorctrl motorctrl.togglebutton1 bit out
newpin motorctrl motorctrl.togglebutton1-not bit out
ready  motorctrl
```

HalRemoteComponent



change detection API: „compiled components“

```
from machinekit import hal

c = hal.Component(cname, mode=TYPE_REMOTE)
c.newpin(„out“, hal.HAL_S32, hal.HAL_OUT)
c.newpin(„in“, hal.HAL_BIT, hal.HAL_IN)
c.ready()

#.. link pins .. and change them
print c.changed()
```

what haltalk does

- remote command/status tap into HAL
- dual-faced: HAL C API
- zeromq/protobuf/discovery
- serves HAL groups and remote components by „adopting“ them
- *config service*
- *authentication/encryption per-socket*

adding a message type

- add type tag to types.proto:MsgType

zeroMQ basics

- sockets principles
- message structure
- communication patterns
  - event notify, RPC: ROUTER/DEALER
  - state replication: XSUB/XPUB

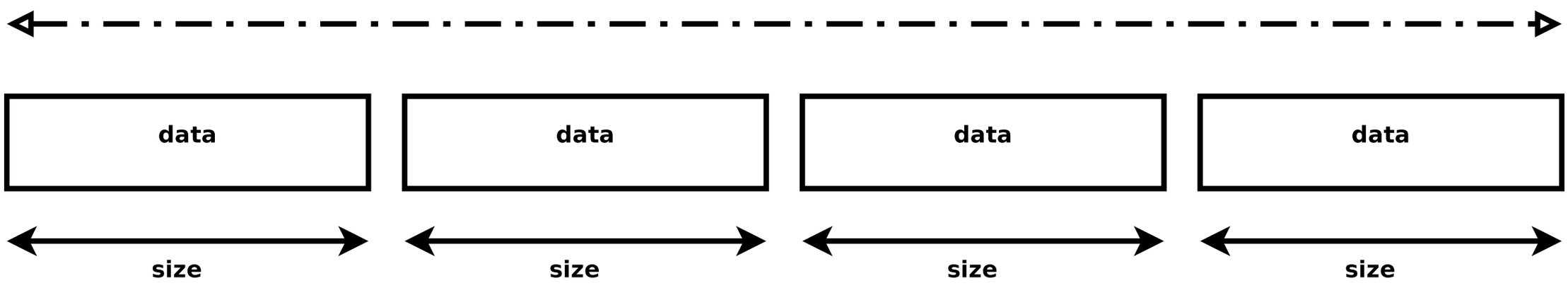
zeroMQ socket principles

- identified by URI: „tcp://1.2.3.4:5500“
- several transports (tcp, ipc, inproc, pgm)
- actually an **array** of connections
- no startup order dependencies
- automatic reconnect
- typed **-** implies a **communications pattern**
- **identity -** a string

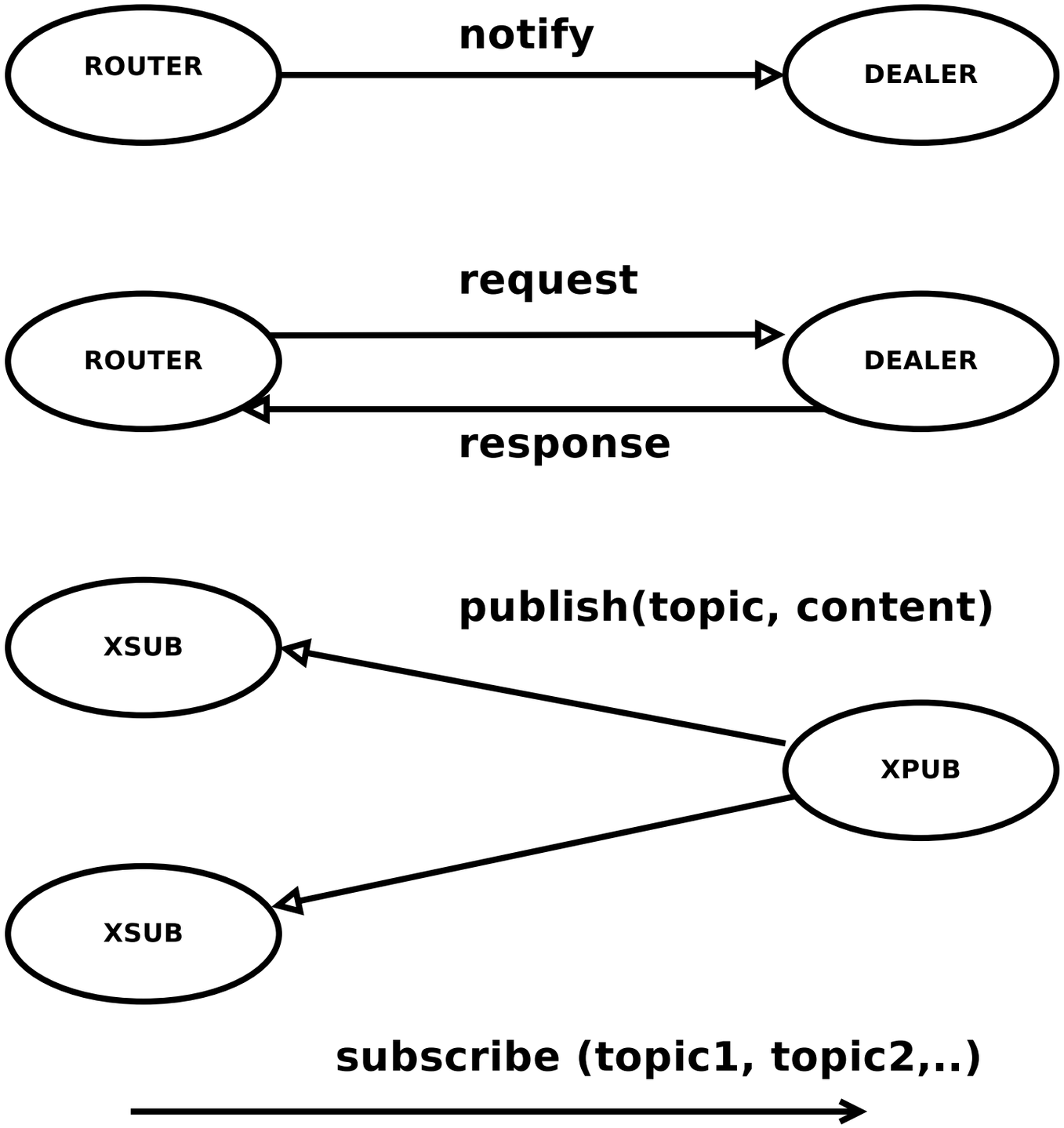
zeroMQ message structure

frame

multipart message



zeroMQ patterns



request/reply client

```
import zmq
import time
import os

context = zmq.Context()
socket = context.socket(zmq.DEALER)
socket.identity = "client-pid%d" % os.getpid()
socket.connect("tcp://127.0.0.1:5700")

for i in range(100):
    socket.send("request %d" % i)
    print socket.recv()
    time.sleep(1)
```

request/reply server

```
import zmq
import time
import sys

context = zmq.Context()

socket = context.socket(zmq.ROUTER)
socket.bind("tcp://127.0.0.1:5700")

count = 0
while True:
    req = socket.recv_multipart()
    print "got: ", req
    socket.send_multipart([req[0], "reply to %s" % req[1]])
```

subscriber

```
import zmq
import time
import sys
context = zmq.Context()
socket = context.socket(zmq.XSUB)
socket.connect("tcp://127.0.0.1:5500")

if len(sys.argv) > 1:
    for topic in sys.argv[1:]:
        print "subscribing to:", topic
        socket.send("\001%s" % topic)
else:
    print "subscribing to all topics"
    socket.send("\001")
while True:
    msg = socket.recv_multipart()
    print "got: ", msg
```

publisher

```
import zmq
import time
import sys
context = zmq.Context()
socket = context.socket(zmq.XPUB)
socket.setsockopt(zmq.XPUB_VERBOSE, 1)
socket.bind("tcp://127.0.0.1:5500")
p = zmq.Poller()
p.register(socket, zmq.POLLIN)
timeout = 1000
count = 0
while True:
    events = dict(p.poll(timeout))
    if socket in events and events[socket] == zmq.POLLIN:
        rx = socket.recv()
        if rx[0] == '\001':
            print "subscribe event for topic: '%s'" % rx[1:]
        if rx[0] == '\00':
            print "unsubscribe event for topic: '%s'" % rx[1:]

    for topic in sys.argv[1:]:
        socket.send_multipart([topic,"update #%d for topic %s" % (count, topic)])
    count += 1
```

usage of patterns

- PUB/SUB:
  - halrcomp: learn of pin changes
  - halgroup: observe signal changes
  - *interpreter/task/iocontrol: status updates*
- REQ/REPLY:
  - set a pin value
  - *start a program run*
  - *change a tool*

state replication pattern

- enables a client to track a blob of state held in another server entity
- identified by a topic (compname, task,interp..)
- used with PUB/SUB sockets
- on initial connect (SUBSCRIBE), server sends a full description of state
- on state change: send incremental updates

protobuf 101: describing messages

// C:

```
struct command {
  int type;
  double b;
  char msg[255];
  ...
  }
```

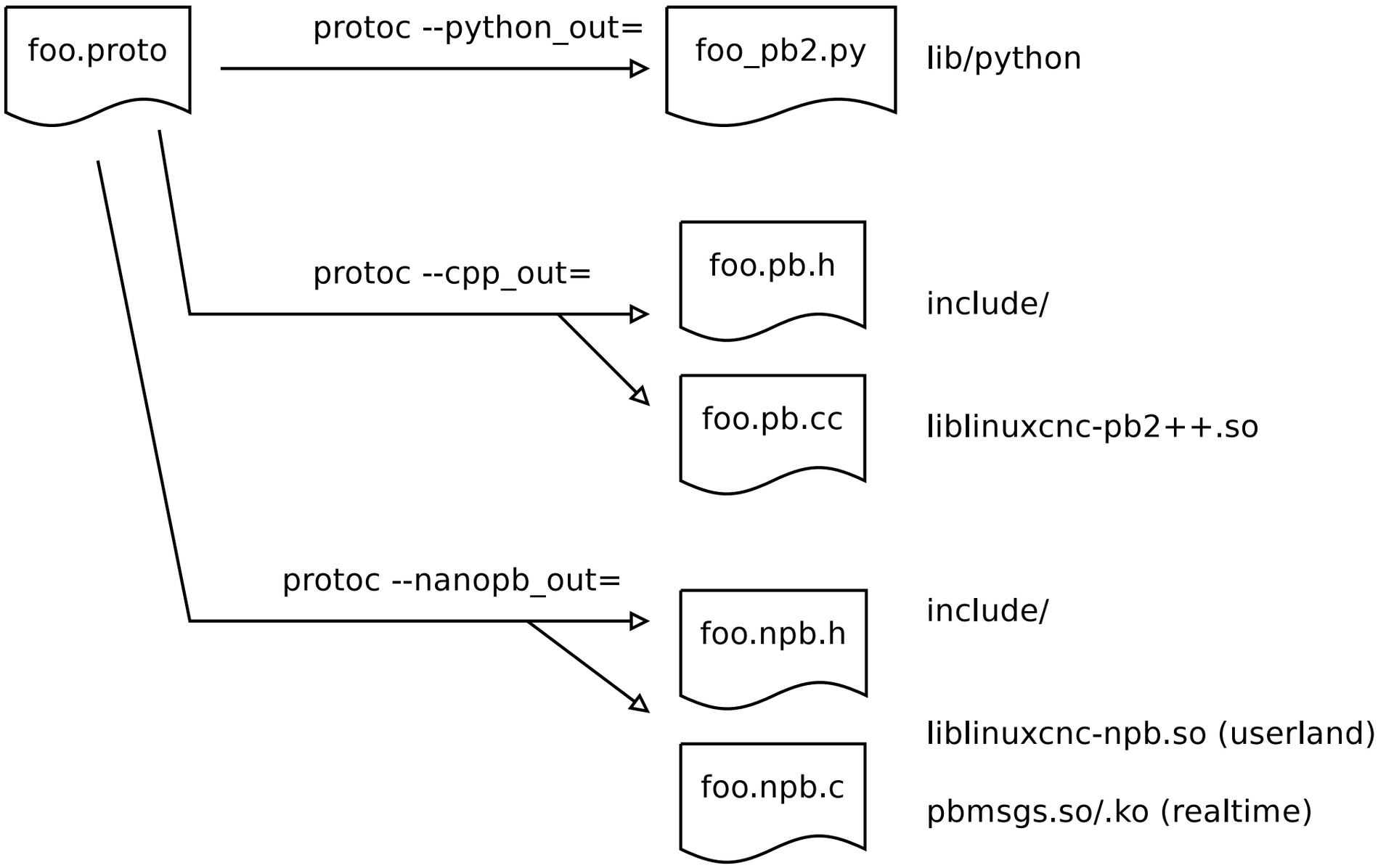
// protobuf

```
message command {
    required  int32 type = 1;
    required  double b = 2;
    required string msg = 3;

    optional double velocity = 5;
    repeated Segment segment = 6;

}
```

what protobuf is

- message description format (.proto)
- scalars, strings, nested messages
- optional and repeated fields
- portable wire format
- compiled into **bindings** (40+ languages)
- linked into application + runtime support
- **introspection** (used in JSON translation)
- now using: C++, Python .. and C (nanopb)
- wireline <-> TextFormat <-> JSON, any direction

the protobuf build process



more „real“ proto definition

```
package pb;
import "nanopb.proto"; // for RT options
enum DemoType {
    DT_POLYLINE      = 1;
    DT_ABORT         = 2;
    DT_GETPOSITION   = 3;
    DT_STATUS        = 4;
    DT_ERROR         = 5;
}
message Target {
    optional double x = 1;
    optional double y = 2;
    optional double z = 3;
};
enum SegmentType {
    ST_LINE = 1;
    ST_ARC  = 2;
}
message Segment {
    required  SegmentType type = 1;
    optional  Target      end = 2;
    optional  Target      center = 3;   // DT_ARC
    optional  Target      normal = 4;   // DT_ARC

    // optional vel/acc overrides per segment:
    optional  double      velocity = 5;
    optional  double      acceleration = 6;
}
```

```
message DemoContainer {
    required  DemoType type = 1;
    repeated Segment segment = 2  [(nanopb).max_count = 20];

    // default vel/acc for whole polyline
    optional double velocity = 5;
    optional double acceleration = 6;

    // if used as reply from RT:
    optional Target current = 7;
    optional string note = 8 [(nanopb).max_size = 30];
}
```

usage example

```
import json  # for pretty printer
import binascii
import
google.protobuf.text_format

# create a message object:
d = DemoContainer()
# fill in what's required
d.type = DT_POLYLINE
d.velocity = 50
d.acceleration = 500

s = d.segment.add()
s.type = ST_LINE
s.end.x = 100
s.end.y = 200

s = d.segment.add()
s.type = ST_ARC
s.end.x = 50
s.end.y = 80
s.end.z = 0

s.center.x = 120
s.center.y = 150
s.center.z = 0

s.normal.x = 0
s.normal.y = 0
s.normal.z = 1
```

```
s = d.segment.add()
s.type = ST_LINE
s.end.x = 0.0
s.end.z = 10.0
s.velocity = 30.0
s.acceleration = 200.0

if sys.argv[1] == "text":
    print str(d)

buffer = d.SerializeToString()
size =  d.ByteSize()

if sys.argv[1] == "binary":
    os.write(1,str(buffer))

if sys.argv[1] == "hex":
    print size,  binascii.hexlify(buffer)

if sys.argv[1] == "json":
    jsonout = d.SerializeToJSON()
    print json.dumps(json.loads(jsonout), indent=4)
```

wireline format

```
$ python pydemo.py hex
177
080112160801121209000000000000059401100000000000
006940125908021121b090000000000049401100000000
00005440190000000000000001a1b0900000000005e
401100000000c062401900000000000000221b0900
000000000000011000000000000001900000000000f
03f1228080112120900000000000000001900000000000
2440290000000000003e4031000000000006940290000
0000000049403100000000407f40
```

text format

```
$ python pydemo.py text
type: DT_POLYLINE
segment {
  type: ST_LINE
  end {
    x: 100
    y: 200
  }
}
segment {
  type: ST_ARC
  end {
    x: 50
    y: 80
    z: 0
  }
  center {
    x: 120
    y: 150
    z: 0
  }
  normal {
    x: 0
    y: 0
    z: 1
  }
}
```

```
segment {
  type: ST_LINE
  end {
    x: 0.0
    z: 10.0
  }
  velocity: 30.0
  acceleration: 200.0
}
velocity: 50
acceleration: 500
```

JSON format

```
$ python pydemo.py json
{
    "acceleration": 500,
    "type": 1,
    "segment": [
        {
            "type": 1,
            "end": {
                "y": 200,
                "x": 100
            }
        },
        {
            "type": 2,
            "end": {
                "y": 80,
                "x": 50,
                "z": 0
            },
            "center": {
                "y": 150,
                "x": 120,
                "z": 0
            },
            "normal": {
                "y": 0,
                "x": 0,
                "z": 1
            }
        },
        {
            "acceleration": 200.0,
            "type": 1,
            "end": {
                "x": 0.0,
                "z": 10.0
            },
            "velocity": 30.0
        }
    ],
    "velocity": 50
}
```

webtalk: the web hook

- tiny http + Websockets server
- host side talks zeroMQ/protobuf
- client side talks JSON/websocket
- connect URI tells server what to do

webtalk example (py):

```
dest = "tcp://127.0.0.1:5700"
ws = websocket.WebSocketApp("ws://127.0.0.1:7681/?"
                            "debug=-1&"
                            "connect=%s&"
                            "type=dealer&"
                            "policy=demo&"
                            "identity=%s" %
                            (dest, identity),
                            on_message = on_message,
                            on_open = on_open,
                            on_error = on_error,
                            on_close = on_close)
```

work through live examples..

status

- HAL „remotification“ functionally complete
- TBD:
  - authentication/credentials per service
  - *config service*
  - *messagebus*
- usable now for remote **HAL** UI‘s
- gladevcp, QtQuickVCP: done - work remotely
- build integration - done

tutorial examples

- https://github.com/mhaberler/machinekit/tree/machinetalk-preview-2/src/machinetalk/tutorial
- basics of protobuf, zeromq, ringbuffers
- complete end-to-end example spanning JSON/websockets ... realtime component and back