`linr.c` is a lightweight computation server (a dozen hundred lines of POSIX C code) which is able to exploit multiprocessor hardware in a natural way. Use for example `gcc -O3 -lrt linr.c -o linr` to compile it.

1 Invocation

When invoked from the command-line with `./linr engine socket`, it will listen on the chosen `socket` and expects to be fed with graph computation tasks which will be reduced according to rules contained in the rewriting `engine`. Results will progressively be sent back to the client using the same graph-transmission protocol that was used for the input.

Additional command-line options, with default values:

```
-p=4      number of worker processes
-m=2048   upper memory limit, in KiB
-c=512    network connection limit
-b=4096   network buffers size
-q=16     network listen queue size
```

2 Memory resources

A main shared memory segment is initially created to host computation. It will be populated and formatted lazily as a pool of slots, which are typically 8, 16 or 32 bytes blocks that can store up to four addresses that reference other slots. Those slots are mainly used to store nodes, but they also conveniently provide interaction emplacements where new redexes can spawn.

A second small shared memory segment is used to host the chosen engine bytecode.

Last, a small data structure and a pair of input and output buffers is mallocated by the main (communication-handling) process upon each client connection.

3 Computational resources

In order to exploit available computing resources, side unix processes (worker processes) are created. One is needed for each CPU core to be used. A semaphore is used to wake up worker processes as needed.

The rudimentary synchronization needed to allow proper combination of the graph-rewriting steps performed by worker processes relies on a simple use of `__sync_test_and_set` instructions. Occasional accesses to shared facilities (e.g. the global pools of free-slots and redex-slots) still rely on global locking mechanisms.

4 Communication protocol

Graphs which are considered here both as computation tasks and computation results are just sets of labeled nodes (labels are encoded as integers) which may be connected to up to four other nodes through wires.

For the time of the network transmission, arbitrary integer identifiers are associated to each wire. Those wire identifiers can be reused as soon as both nodes it connects have been transmitted. As a convention, since communication is bidirectional and in order to prevent identifier collisions, the client must only use even integers when allocating new wire identifiers, and the server will use odd integers.

Nodes are transmitted sequentially (but the ordering can be chosen arbitrarily) using byte sequences. The first byte of each node sequence defines its total length (which is expected to be small, typically less than 10, and currently accepted up to 64). The following bytes of the sequence encode successive integers: the number associated to the node label (non null integer), the identifier of the wire connected to its principal port, the identifier of the wire connected to its 1st auxiliary port (if any), the identifier of the wire connected to its 2nd auxiliary port (if any), etc... Handling more than 3 auxiliary ports is not currently implemented.

Each integer n is itself encoded as a sequence of bytes in the following way: any $n < 128$ is encoded within a single byte in the standard way. otherwise it is encoded as byte $128 + n \bmod 128$, followed by the sequence which encodes $n/128$. For example, a node with label 4, whose principal port is connected to wire 34 and whose 1st and 2nd auxiliary ports are linked to wires 82 and 265, will be transmitted using the following byte sequence: `6-4-34-82-137-2`.

A special node label 0 is used:

- followed by byte 0 and two integer identifiers, to link together two wires that have already been assigned different identifiers.
- (by the client only) followed by byte 1 and two integer identifiers, to indicate that no further input will mention the second identifier, instead the server is asked to send the output relevant to this link to the provided first identifier.

5 Rewriting engines

The bytecode engine format is still not definitive, a compiled version of the following set of nodes and reduction rules, which implements basic linear logic with recursion, is provided.

```
1:one 2:ten 3:bottom 4:par
5:weak 6:contr 7:derel 8:prom-out 9:prom-in
10:epsilon_ 11:delta_ 12:omicron_
13:lplus 14:rplus 15:with 16:switch
17:lambda_ 18:rho_
19:rec
20:uncast 21:brcast

one() -- bottom()
ten(x, y) -- par(x, y)
weak() -- prom-out(epsilon0(), epsilon_())
derel(x) -- prom-out(omicron0(x), omicron_())
contr(prom-out(x, cx), prom-out(y, cy)) --
  prom-out(delta0(x, y), delta_(cx, cy))
prom-in(epsilon0(), weak()) -- epsilon_()
prom-in(omicron0(z), z) -- omicron_()
prom-in(delta0(zx, zy), contr(cx, cy)) --
  delta_(prom-in(zx, cx), prom-in(zy, cy))
lplus(z) -- with(lambda0(z), kappa0(), lambda_())
```

```
rplus(z) -- with(kappa0(), rho0(z), rho_())
lambda_() -- switch(lambda0(z), kappa0(), z)
rho_() -- switch(kappa0(), rho0(z), z)
weak() -- rec(epsilon0(), epsilon0(), epsilon_())
derel(x) -- rec(delta0(omicron0(x), y),
  delta0(omicron0(rec(y, ry, cy)), ry),
  delta_(omicron_(), cy))
contr(rec(x, rx, cx), rec(y, ry, cy)) --
  rec(delta0(x, y), delta0(rx, ry), delta_(cx, cy))
uncast() -- epsilon_()
uncast() -- delta_(uncast(), uncast())
uncast() -- omicron_()
uncast() -- lambda_()
uncast() -- rho_()
brcast(epsilon_(), epsilon_()) -- epsilon_()
brcast(delta_(xx, yx), delta_(xy, yy)) --
  delta_(brcast(xx, xy), brcast(yx, yy))
brcast(omicron_(), omicron_()) -- omicron_()
brcast(lambda_(), lambda_()) -- lambda_()
brcast(rho_(), rho_()) -- rho_()
```