JSON-based configuration of kernel subsystems

Giuseppe Lettieri, Luigi Rizzo
Università di Pisa

BSDCan 2016, Ottawa

kernel configuration

- Setting global parameters
- Creating new objects
  - with parameters
  - with unique names
- Deleting objects
- Setting per-object parameters
- Connecting/disconnecting objects

The netmap example

- ports
  - some of them may be created on demand (pipes, monitors, emulated mode ports, passthrough ports, …)
  - several parameters
- memory regions
  - several parameters
  - "private" memory regions are created on demand
- bridges
  - **only** created on demand
- ports need to be bound to memory regions
- ports may be connected to bridges

netmap limitations

- port to memory region binding fixed by port type
- hardware ports all bind to the global memory region
- private memory regions and bridges are on-demand only
- no nice way to set parameters for on-demand objects

The available choices

- Ad-hoc system calls (and tools)
- ioctl()
- sysctl()
- pseudo-devices (and maybe tools)
- pseudo-filesystems

Limitations

- the specialized ones are diverse and ad-hoc
- the general ones are not atomic
- pseudo-filesystems are very complex
  - and still not atomic!

JSON

- Use JSON
  - (potentially) general
  - accessible from almost any language
  - it may express atomicity via groupings ({} and [])

User Interaction

- `read()`/`write()` **from/to a special device**
- `read()`**: get a JSON representation of the subsystem objects**
- `write()`**: push a JSON "template" for queries and/or updates**

Example: read()

```
# cat /dev/netmap
{
  “port”: {
    “em0”: {
      “memid”: “1”,
      …
    }
  },
  “mem”: {
    “1”: {
      …
    }
  }
}
```

Example:write()

updating:

```
# echo '{"port":{"em0":{"memid": "2"}}}'\
  > /dev/netmap
# cat /dev/netmap
{
   "port": {
     "em0": {
       "memid": "2",
   …
}
```

atomicity

```
# echo '{"mem": {"1": {"req_buf": \
   { "size": 9000, "num": 50000 }}}}' >/dev/netmap
```

The parser internally takes locks:

```
{
```

lock netmap ctrl

```
  "mem": {
    "1" : {
```

lock mem-object "1"

```
      "req_buf": {
        "size": 9000,
        "num": 50000
      }
```

unlock mem-object "1"

```
    }
  }
}
```

unlock netmap ctrl

Human friendly syntax:

bare words:

```
# echo '{port:{em0:{memid: "2"}}}' \
   > /dev/netmap
```

"dot" substitution:

"`.X`" becomes "`{X}`" or "`:{X}`" as needed:

```
# echo '.port.em0.memid:"2"' > /dev/netmap
```

number to string conversion:

```
# echo .port.em0.memid:2 > /dev/netmap

# echo .mem.1.req_buf:{size:9000, num:5000} \
    >/dev/netmap
```

The protocol

- The parsing of what you `write()` starts when:
  - you `close()` the file descriptor, or
  - you start `read()`ing
- By `read()`ing you get the reply to your last action
  - same shape as the input
  - updates show the result (may be an error)
  - “queries” are filled

example: write(); read()

We need a “tool”:

```
# cat /usr/sbin/nmconf
exec 3<>/dev/netmap
cat “$@“ >&3
cat <&3
```

example: nmconf

```
# nmconf
{
  "port": {
    "em0": {
      "memid": "?"
    }
  }
}
```

ctrl+D

```
{
  "port": {
    "em0": {
      "memid": "2"
    }
  }
}
```

Creating Objects

```
# nmconf
{
  "mem": {
    "&": {
      "req_buf": { … }
    }
  }
}
ctrl+D
{
  "mem": {
    "3": {
        …
    }
  }
}
```

funny character

unique name

Connecting objects

```
# cat <<EOF >/dev/netmap
{
  “mem”: {
    “&X”: {
     …
    }
  }
  “port”: {
    “em0”: {
      “memid”: “$X”
    }
  }
}
EOF
```

assign variable

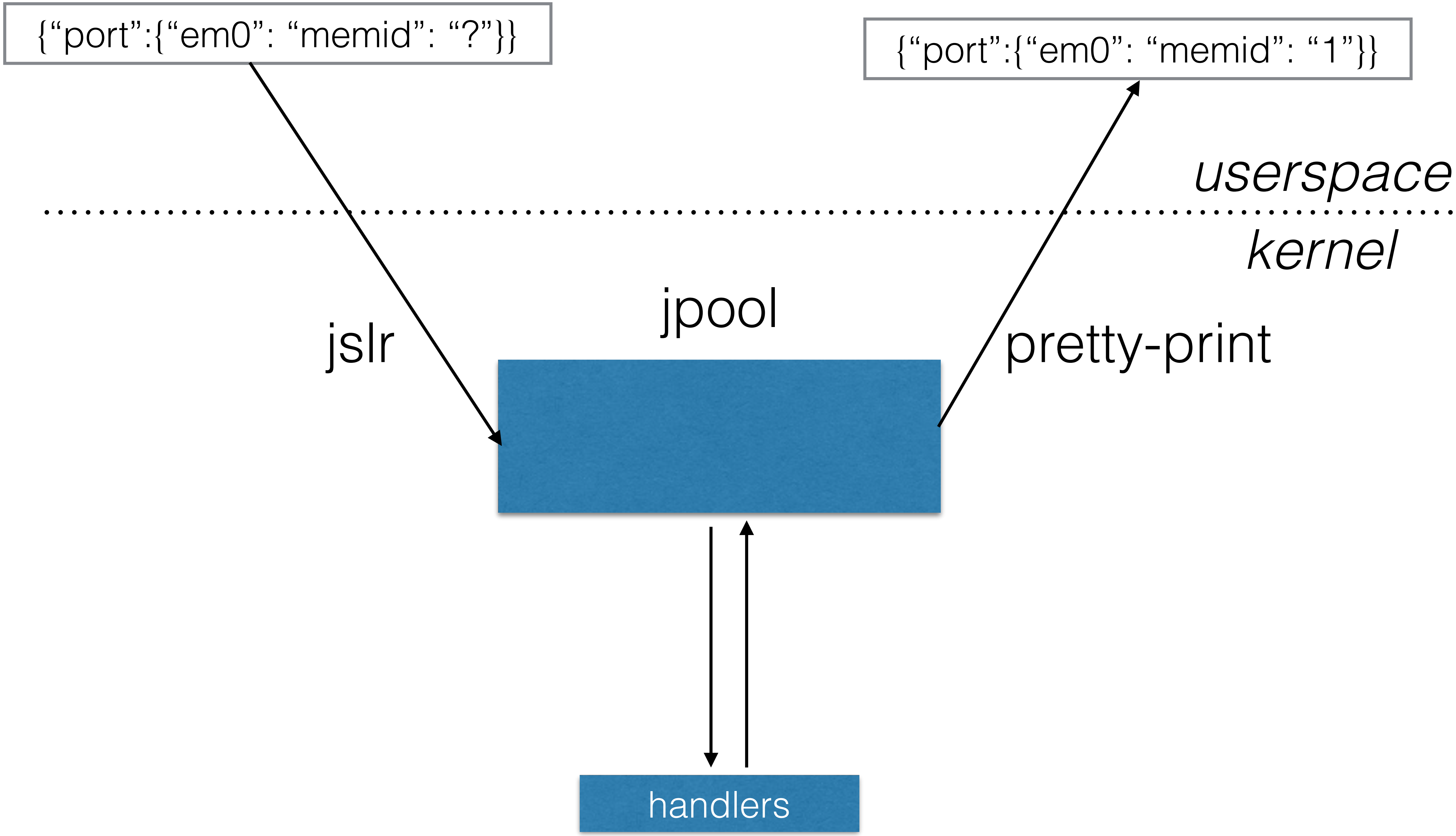
replace variable

Variables

```
# cat <<EOF >/dev/netmap
{
  "port": {
    "em0": {
      "memid": "?X"
    }
    "em1": {
      "memid": "$X"
  }
}
EOF
```

query and assign

Internals


{"port":{"em0": "memid": "?"}}
{"port":{"em0": "memid": "1"}}
userspace
kernel
jpool
jslr
pretty-print
handlers

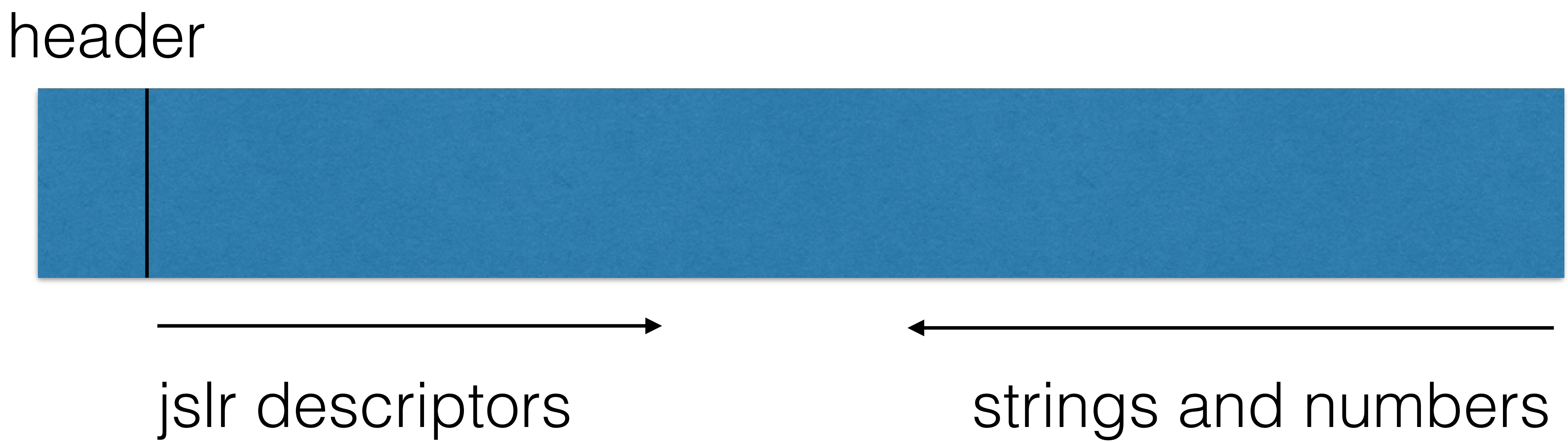
JSLR

- tiny JSON parser
  - ~700 loc, including comments

- optional extensions:
  - bare words
  - dot transformation

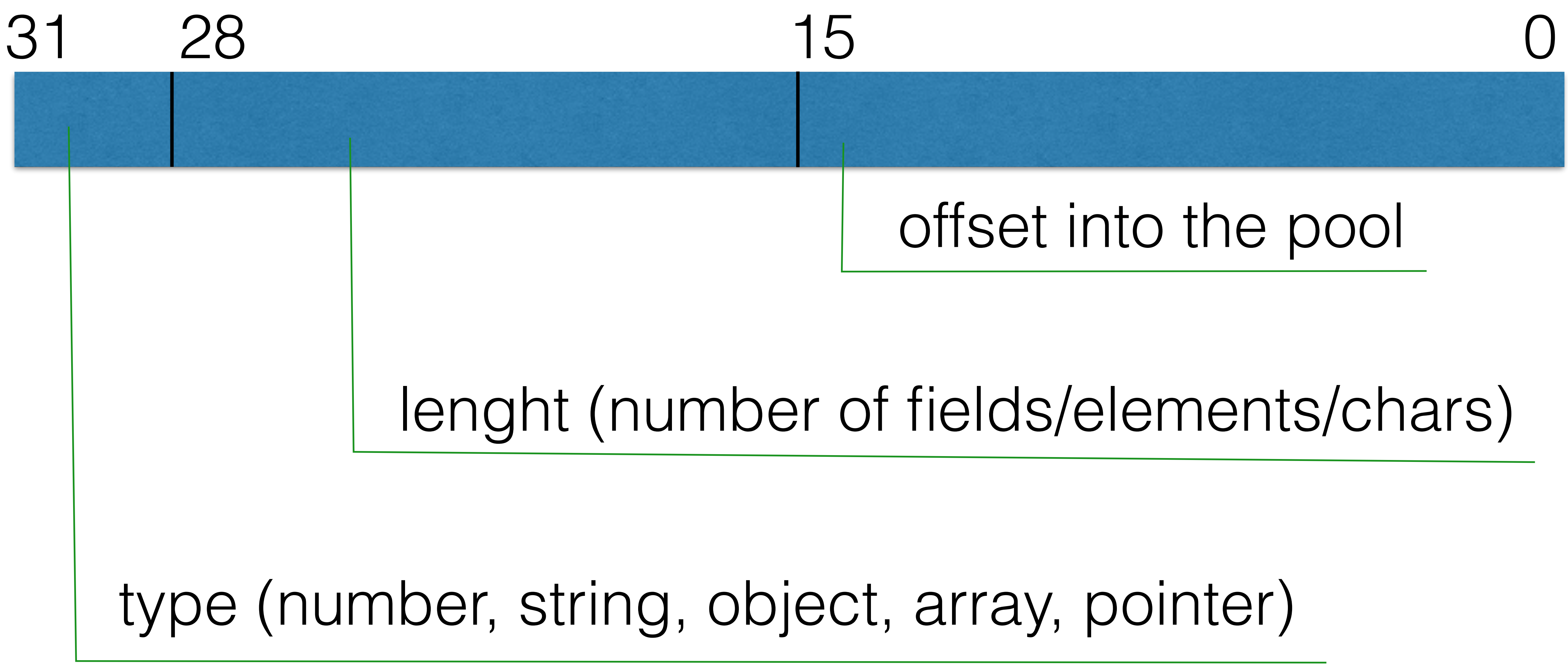
- simplified memory management
  - pool allocated when parsing begins
  - JSON object are created only inside the pool
  - the pool is disposed when the output is produced

JSON pool



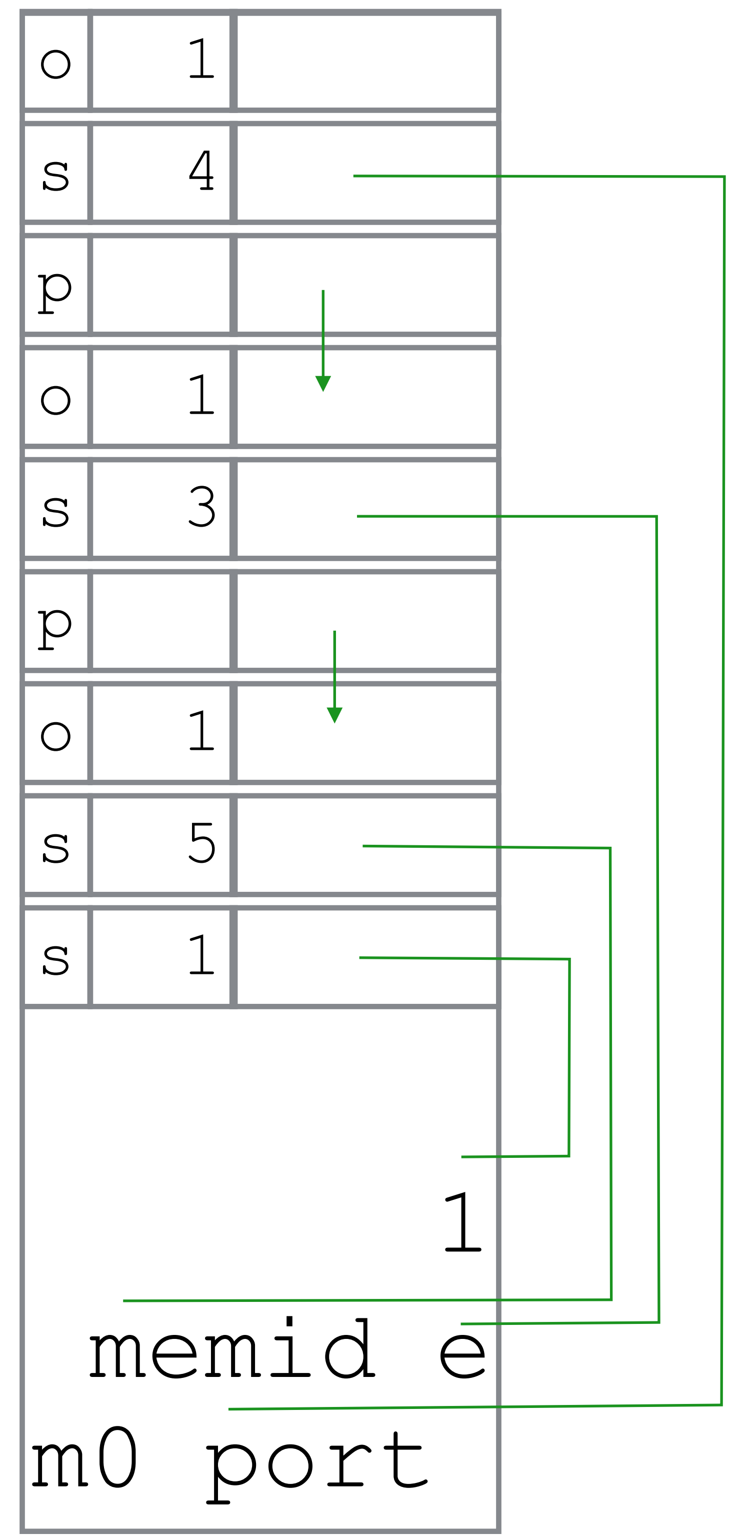
jslr descriptors

```
struct _jpo { … };
```



jslr example

```
{
  "port": {
    "em0": {
      "memid": "1"
    }
  }
}
```



Handlers

- Responsible for interpreting and producing JSON
- organized hierarchically starting at a root handler
- Polymorphic structures, three callbacks:
  - `interp`: given a _jpo, apply the updates, return new _jpo
  - `dump`: return a _jpo describing the below hierarchy
  - `bracket`: called when entering/leaving the hierarchy
- The _jpo returned by jslr parsing is passed to the root handler; the returned _jpo is pretty-printed to obtain the output JSON

Example handers

- `nm_jp_num`: update/retrieve numbers
- `nm_jp_dict`: dictionary: maps strings to other handlers; if dumped show up as '{ … }'
- E.g., for /dev/netmap
  - root handler: a dict mapping "mem" and "port" to handlers
  - the "mem" handler is a dict mapping memory areas id to other handlers
  - likewise, the "port" is yet another dict mapping port names to other handlers

Handlers for C objects

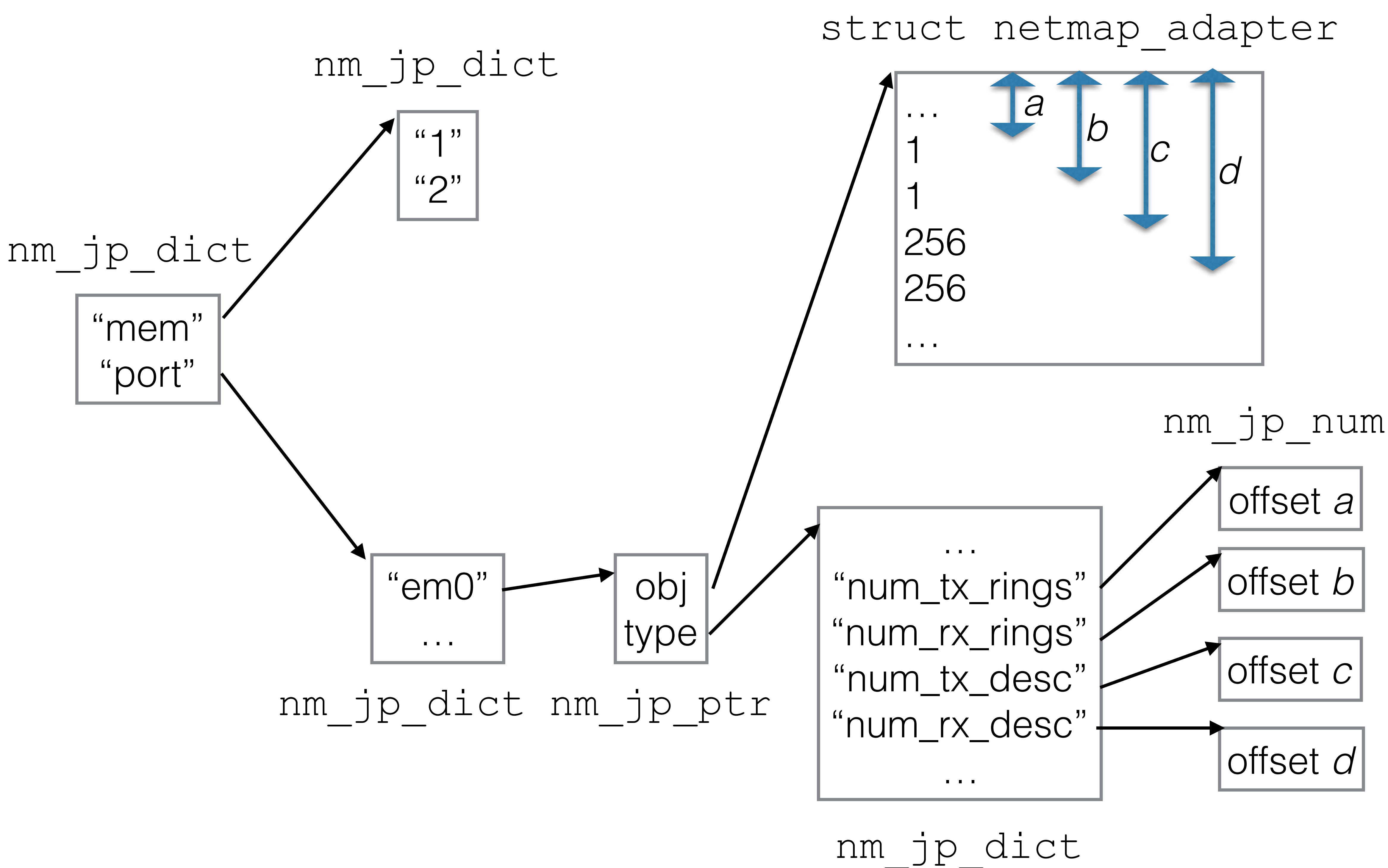
- retrive/update values from existing C objects
- parsing is in the context of a "current C object"
- `nm_jp_ptr` handlers: change/restore the current C object and pass control to the proper "C-type" handler
- if the C object is a struct instance, the C-type handler is typically a dict mapping field names to their handlers

Handlers example

```
struct netmap_adapter {
  …
  u_int num_tx_rings;
  u_int num_rx_rings;
  u_int num_tx_desc;
  u_int num_rx_desc;
  …
};
```

```
{
  “port”: {
    “em0”: {
      …
      “num_tx_rings”: 1,
      “num_rx_rings”: 1,
      “num_tx_desc”: 256,
      “num_rx_desc”: 256,
      …
    }
  }
}
```

Hierarchy example



Helper macros

```
NM_JPO_DECLARE_CLASS(port, struct netmap_adapter)
   …
   NM_JPO_RONUM(port, num_tx_rings)
   NM_JPO_RONUM(port, num_rx_rings)
   NM_JPO_RONUM(port, num_tx_desc)
   NM_JPO_RONUM(port, num_rx_desc)
   …
NM_JPO_CLASS_END(port, port_bracket)
```

`NM_JPO_CLASS(port)` is then an `nm_jp_dict` that describes `struct netmap_adapter`s

Future work

- **What to do with “[…]”?**

- **Implement searches (related to the above)?**

Thank you!