BSD Sockets API in Zephyr RTOS

Paul Sokolovsky
Linaro LITE
Short introduction to Zephyr RTOS

- Open Source project under auspices of Linux Foundation, so “Linux’ little brother”
- Apache 2.0 license
- IoT-targeted, thus strong communications and networking subsystems
- Undergoes extensive growth
- Needs more eyes and hands to make “bugs shallow” and cover gaps in the functionality
Introduction to Networking in Zephyr

- Native (custom-written) IP stack
- IPv6- and 6LoWPAN-enabled from the start
- Strong emphasis on integration of different technologies (IPv4 and IPv6 at the same time, SLIP, Ethernet, BLE, 802.15.4 bearers, multiple bearers, etc.)
- Optimized for low memory usage
- Has growing library of application protocol implementations
- Not yet fully ready and still has some issues (needs more hands and eyes).
Details of native Zephyr networking API

- Connection endpoint is represented by “network context”, which is, effectively, a socket
- Data/event delivery is callback-based and packet-based
- Packet consists of linked list of “data fragments”, i.e. data is not contiguous
- Despite these (big) differences, other semantic and behavioral details are inspired/aligned with BSD Sockets
Complications when using native IP stack

- Callback-based API, callback hell is just few steps away
- For UDP, packet-based API is fine, but non-contiguous buffers are still a nuisance. Many apps will want to linearize buffers. Another alternative Zephyr offers is adhoc API to “read” values of particular types.
- With TCP which is stream-based, situation is much worse, as packets make a bad abstraction to work with it. Correctly working apps would want not just linearize packets, but buffer across packets, as indeed, HTTP’s “GET /foobar.html” can be split among packets as “G”, “E”, “T /foo”, “bar.htm”, “l”. But it’s easy to “overlook” that and write incorrect TCP apps based on the assumption that a packet (or particular data fragment) contains complete element of higher-level protocol.
Issue examples with native IP stack

- ZEP-2362 “Multiple issues with http_server library design and implementation”
  - Manually with "telnet 2001:db8::1 80", server sends response immediately after the initial request line, e.g. "GET /index.html HTTP/1.0".
  - [https://github.com/zephyrproject-rtos/zephyr/pull/802](https://github.com/zephyrproject-rtos/zephyr/pull/802) “Use net_context_put(), not_context_unref().”

- samples/net/irc_bot: /* TODO: handle messages that spans multiple packets? */

- [https://github.com/zephyrproject-rtos/zephyr/pull/1374](https://github.com/zephyrproject-rtos/zephyr/pull/1374) “Support SNTP client library”, review comments:
  - “this looks like copy-paste from bsd socket code”
  - “Again, code duplicate from bsd sockets.”
  - “FIFO may contain more packets, which are now leaked. Saving app writers from mistakes like this was another reason for developing bsd sockets API.”
  - “The original socket-based sample code is easy and compact. I don’t need to handle FIFO or semaphore. It just works.”
Why add BSD Sockets?

- Our stakeholders told us (d’oh)
- Native API needs learning, while everyone knows BSD Sockets
- Native API is cumbersome to use
- Use of native API is prone to bugs and making wrong assumptions

In other words, BSD Sockets offer:

- Ease of use
- Reuse of existing network programming experience
- Rapid prototyping of apps
- Potentially, porting existing code
Requirements

- Should be a **thin** layer on top of native API (no aim to replace native API, users of native API [only] should not be burdened with sockets overhead)
- However, push any suitable changes and fixes down the stack
- Following Zephyr aims, have minimal code size and memory usage
- Of course, implement a subset of BSD Sockets API (not “full API”)
- Still, should be possible to write at least simple “fully portable” apps working also on a real POSIX system (like Linux)
- Achieving minimal sizes, it’s ok to make not fully general assumptions (which may be revisited in the future)
- Likewise, use all the tricks which POSIX allows us to achieve minimal size/overhead, e.g. short reads/short writes
Development strategy

- Initially implement out-of-tree prototype
- Select an app with more or less good BSD Sockets API coverage, and “port” it to Zephyr, by implementing Sockets API functions in terms of native API, collecting knowledge what would need to be pushed where for in-tree implementation
- Meet the contender: MicroPython
  - Already had a basic Zephyr port
  - Has a full-fledged Python sockets API, directly mapping to C API
  - Contains similar implementation of sockets API on top of lwIP native API
Implementation 1/2

- Initial implementation available in Zephyr 1.9
- Fully separate from the core stack, in subsys/net/lib/ (like application protocols)
- Doesn’t use any threads (no stack, etc. overheads)
- The whole idea of the implementation is to do “impedance matching” between push-style, callback-based native API and pull-style BSD Sockets API
- A FIFO was used for this “impedance matching”, a receive callback puts a packet in queue, recv() takes it from there.
- Likewise for new connections and accept(). As a socket can’t be both listening and receive data, a single FIFO is actually used (example of memory use optimization).
- Only UDP recv() takes a whole packet out of FIFO, TCP one actually takes it data fragment by fragment (employing POSIX short reads property)
Implementation 2/2

- Send is largely pass-thru, but unlike native API, takes care to limit data in a packet to MTU (uses short writes).
- Pass-thru essentially means “flood send” approach, which isn’t the only send policy, not friendly to other sockets, to system resources, and actually hits buffer allocation deadlocks in the current Zephyr native stack.
- Alternative approach would be sequential send, but would require a semaphore per socket (extra memory overhead).
- poll() implementation which rewrites struct pollfd array to a static array of Zephyr native poll structures. Doesn’t really support write pollability (POLLOUT), but it can be implemented with the semaphore above.
Status

- Both IPv4 and IPv6 supported
- Both UDP and TCP supported
- Both blocking and non-blocking sockets supported
- `poll()` supported, no `select()`, or `epoll()` (so far at least)
- Assumes that RAM lies in the first half of address space (so a pointer cast to an int is positive)
- Two echo server examples are provided: single-threaded blocking and concurrent asynchronous using non-blocking sockets. Both can be built for POSIX (tested on Linux), `#ifdef`s apply only to headers.
- Needs more samples, d’oh
Known issues and future work

- Many issues are rooted in Zephyr stack known issues
- Solution to a flooding send and write pollability
- Most useful setsockopt() calls (e.g., multicast group joining)
- Needs more samples
- Needs more users and testers in general
- Don’t even ask about performance - the aim is to get networking in Zephyr work right first
Thank You

#SFO17

SFO17 keynotes and videos on: connect.linaro.org
For further information: www.linaro.org