PTMP lessons from ProtoDUNE-SP for DUNE FD

Brett Viren

October 14, 2020

Topics

In the area of **message passing** and based on experience of PTMP at ProtoDUNE-SP:

- Select subset of the **problems/lessons** learned.
- Suggest some **requirements** for new DUNE FD DAQ inter-process messaging (IPM) package to consider.
- Provide **pointers** to some existing solutions.

Problem: sending and receiving messages

Algorithms need non-trivial interface to send/recv messages.

- Had to separate **knowing** a send/recv will succeed from **doing** the send/recv.
- Needed to wait for input on multiple sockets with different semantics.
- PTMP initially tried to dumb this down and with `TPSender` and `TPReceiver`. These simple "sink/source" abstractions proved useless/harmful.

IPM Requirements

- Multi-socket input and output **poller** is needed.
- Ask, "which sockets **ready** for input, output or both"
- The ask must **wait** with a zero, finite or infinite timeout.
- Polling must be independent from actual send/recv operation.

Solutions

`cppzmq` pollers (tour link) provide idiomatic C++ wrapper around `libzmq`'s poller.

Problem: combining message codec and schema

PTMP used single, monolithic message defined with ProtoBuf

- General algorithms become bound to specific ProtoBuf types.
- Generated reticence to adopt novel message types (called for by users),
- Suffered full-message deserialization to read small parts of messages.

IPM Requirements

- Low-level message send/recv interface should not specify a codec nor schema and work with anonymous “byte arrays”.
- Supply utils to build byte arrays, handle single-part codec and schema following the layer encapsulation strategy (eg as used in IP protocols).

Solutions

- cppzmq `message_t` (tour link) and `multipart_t` codec (tour link).
- ZIO message schema (next slide)

ZIO message schema

A general, multi-purpose container message, design influenced by input and discussions from ProtoDUNE DS / self-trigger developers. (more info here)

name	type	bytes
magic	"ZIO"	3
level	"%d"	1
form	(eg "JSON", "TEXT")	4
label	free string	N0
origin	uint64	8
granule	uint64	8
seqno	uint64	8
payload	per form	N1
payload	per form	N2
...		

Three types of message parts (aka frames):
(prefix header) + (coordinate header) + [payloads]

ZIO message schema field details

ZIO **prefix header** is a string which may be used as a ZeroMQ "subscription topic" when PUB/SUB is used and which takes this meaning:

magic	mark this as ZIO message
level	may use to set an application level priority
form	user payload hint to guide serializers
label	arbitrary, application-specified string

ZIO **coordinate header** provides meta info for high-level algorithms.

origin	identify the entity that produced the message.
granule	provide an ordering value for the message
seqno	index this message in a sequence of related messages.

ZIO **message payload** provides for arbitrary app data as bytes.

- App chooses serialization format and methods, if any.
- App may use native `cppzmq` or PyZMQ message construction/query methods.

Problem: socket configuration and its coding

PTMP socket configuration

- Jsonnet/JSON as file format and `nlohmann::json` as C++ object representation.
- User configuration fully determines socket types (PUSH/PULL vs PUB/SUB), transport (`inproc://`, `tcp://`, `ipc://`) and addressing.
- **But**, large portion of a PTMP *component* coding was configuration handling
  - Most of that was related to **socket construction and initialization**
  - Hand-written, tedious, repetitive, error-prone coding.

IPM Requirements

- Abstract the socket construction and initialization from the usage by app code.
- Automate configuration parsing and validating
- Configuration (not framework code) determines socket type, transport and addressing.

Solutions

- Recent, related `appfwk` ccm/command developments.
- ZIO node and port socket abstractions.

Problem: network assembly complexity

PTMP's small network was complex, DUNE FD more so

- PTMP's network graph defined with static configuration files.
- Fully enumerated every sockets' type, transport and address.
  - Had to configure component with job/host level information for itself and its peers.

IPM and CCM requirements and **discussion needed**

A system is needed to manage this complexity and assure validity

- Humans want to express desired network in abstract terms
  - "I want 10 TP finders per APA to talk to one TC finder"
  - And not "let me exhaustively tell thousands of IP addresses and ports"
- Do we want to interpret high level expressions to a **fully static** configuration? Do we want to allow some simplification by introducing an amount of **dynamic discovery**?

Solutions

- Current developments in CCM/appfwk commands provide some solutions.
- For a discovery mechanism based on applying rules to abstract identifiers and automatically allocated addresses, see ZIO peer.

Problem: C++ interface to ZeroMQ

PTMP based on CZMQ, ZIO on cppzmq

- `libzmq` and CZMQ are idiomatic-C, `cppzmq` is idiomatic, modern C++.
  - Both work very well but C-idioms are awkward for those used to idiomatic, modern C++.
- ZeroMQ has a non-trivial API and proper bindings are made with $\mathcal{O}(10)$ FTE-year level of effort. A new API is not to be started lightly.
- An interface that dumbs down ZeroMQ is doomed.

IPM Requirements

- Adopt existing C++ binding, do not reinvent it.
- Focus on mid- or high-level API layers, but do not obscure the low-level.

Solutions

- `cppzmq` "low-level" C++ binding, (my recommendation, more in backups)
- `zmqpp` and ZIO are "high-level" C++ bindings with "batteries included"

Problem: data flow control and back pressure

PTMP used basic ZeroMQ transfer patterns

- Drop "tardy" messages so laggards don't ruin it for everyone.
- Utilized PUB/SUB for TCP (*drop-on-mute*) and PUSH/PULL for inter-thread (*block-on-mute*) as rudimentary flow control: hitting mute state is either lossy or costly.

IPM Requirements

- Allow choice of at least *drop* and *block* on mute behaviors for any given link.
- Allow transport and socket types to be configurable.
- Allow / provide an efficient, lossless data transfer protocol supporting app-level reaction to back pressure.

Solutions

- Flexible socket configuration already covered.
- ZIO's credit-based flow protocol was explicitly designed for high-throughput, efficient data transfers with app-aware back-pressure.

Summary of proposed requirements

- Adopt `cppzmq` and do not hide it from message passing algorithms but DO abstract common patterns (eg, PTMP's `TPFilter`) so high-level algorithms have simple functional interface.
- The message codec, format and schema design is very important. ZIO's layered, general-purpose, container-oriented message schema should be adopted/emulated.
- Abstract the socket construction/initialization from user code but retain ability for configuration to specify a variety of socket patterns and transport. ZIO's `node/port` provides an example.
- Support automatic port selection for `bind()` and automatic discovery for `connect()` to allow rule-based link specifications based on arbitrary socket attributes. ZIO's `node/port/peer` gives example.
- Develop/adopt application-level protocols for large data transfers (to big for one message). ZIO's `flow` protocol is designed specifically for this use.

Links to more info

- ZeroMQ github org and guide
- cppzmq github repo and tour
- PTMP github repo
- ZIO github repo and home page

FIN

cppzmq

https://github.com/zeromq/cppzmq

- Idiomatic, "thin" and modern C++ low-level binding to ZeroMQ.
- Header only, can "vendor" or install with CMake support
  - zmq.hpp message, context, buffer, socket, monitor, poller
  - zmq_addon.hpp adds multipart message and simple buffer interface.
- **Hermetic binding** but app may access underlying libzmq if needed.
- Active, meticulous and caring developers
  - (I have first-hand knowledge from contributing some minor things)
- cppzmq headers plus base ZeroMQ docs sufficient but there is also a more wordy cppzmq "tour" document.

I recommend IPM select cppzmq as a base instead of trying to reimplement it.

Some ZIO features

- Began so as to provide generalized "PTMP-NG++"
- High-level binding to ZeroMQ with **batteries included**.
  - Specifically target commonalities in DAQ and HPC applications
- Toolkit paradigm with both C++ (cppzmq) and Python (PyZMQ) APIs
- Mentioned already: **message**, **node**, **port** and **peer** abstractions
- High-level application protocols:
  - **flow** credit-based data streaming (protocol docs) point to point or via a broker. Includes implementation for distribute HDF5-based I/O.
  - **domo** broker-based dynamic client/worker pattern for service-oriented task requests.
- Successfully inter-operates with others
  - distributed HDF5 I/O with Wire-Cell Toolkit
- More docs at the ZIO home page and also ZIO doxygen

ZIO/Zyre discovery mechanism

ZIO uses the Zyre library to support ZeroMQ ZRE protocol.

- New peer broadcasts UDP "greeting" with peer address.
- Existing peer gets greeting connects to address and sends own "greeting".
- Peers now know each other, exchange heartbeats to confirm liveliness.
- Greeting holds peer "name" and arbitrary key/value "headers" to provide peer semantics.

ZIO provides `peer` abstraction, works with `node` and `port`:

- `bind()` sockets, addresses may be dynamic/automatic.
- announce actual bound addresses as ZRE greeting "headers".
- `port connect()` may be specified abstractly, `peer` collects rules
- resolve rules against other `peer` greeting headers to get concrete addresses.
- `peer` finally connects ports to concrete addresses

Allows automatic self-constructing networks based on abstract rules.
App may extend headers to solve novel problems.
App may enact behavior on `peer` loss to resilience to the network.