UDP vs. basic IP

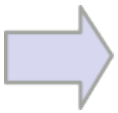
Use reinforced libraries!

High performance parallel programming is hard stuff!

- Patterns like thread-per-connection are very inefficient!
- Patterns like proactor (thread per core) are complex to implement!

Don't reinvent the wheel! Use UDP/TCP via popular libraries.

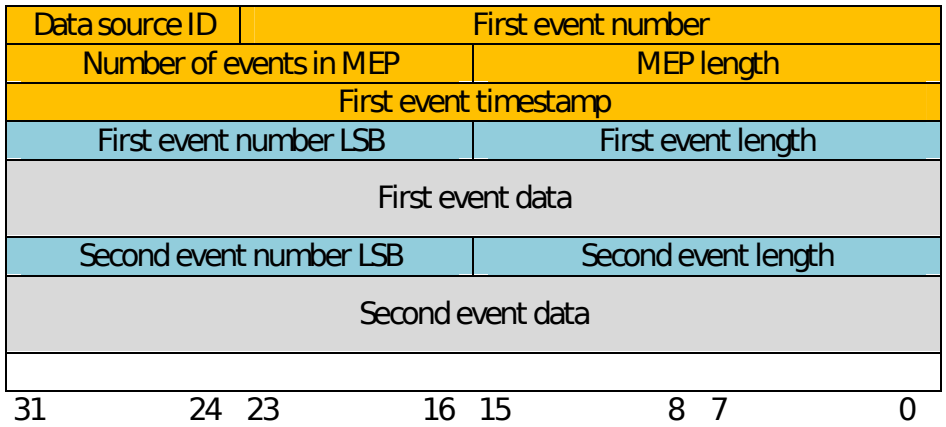
Libs like boost::asio make live easier and "boost" the performance. Using basic IP will make it impossible to use all the functionality of those libraries.



As long as low latency is not of high importance, I would stick to reinforced internet standards like UDP and TCP and implement those with external libraries like boost::asio!

Little endian simplifies software

But performance costs of ntohl only one clock ($\approx 0.33ns$)




```
1 socket.receive(data);
2 // No ntohl needed:
3 memcpy(&firstEventNum,&data[0], 3);
4 memcpy(&sourceID,&data[3], 1);
5 memcpy(&mepLength,&data[4], 2);
6 memcpy(&eventCount,&data[6], 2);
7 memcpy(&firstTimestamp,&data[8], 4);
```

Backup

GWT: An application-like website

Monitoring via your web browser

Every L1/L2 machine writes it's statistics data to one central MySQL server. This data will be summarized via GWT.

Google Web Toolkit

- Mainly a Java to JavaScript compiler
- Duplex communication between client and server possible
- No HTML, PHP/Ruby, Javascript, CSS knowledge needed!
- Only Java (and SQL)!!!

Feels like a locale application without X11-tunneling, program download... **just a standard web browser needed**

What should be displayed

Level 0

- Trigger rate and distribution
- MEP rate to L1
- Data rate to L1

Level 1/2

- Packet loss rate
- Broken event rate
- Processing time of each level
- Load (Memory, CPU)
- Trigger rates and distribution (L1 & L2)
- Data rates to persistent memories

Coincidence with bursts

I'd like to show most of the data in relation to the burst number.

How do I get the burst number corresponding to the event numbers?!

If L0TP is a PC I would let it write this data to the "Burst" table of my monitoring database (*_ID are foreign keys):

ID FirstEvent_ID LastEvent_ID

boost::asio

Why I want to use boost::asio

- Less code
- More readable code
- Proactor design pattern
- Don't want to reinvent the wheel!

Proactor design pattern (Think asynchronous!)

- Almost no synchronization needed
- Thread per core, not per connection
- Less context switching $\Rightarrow$ better performance

Sample program

Proactor design pattern: Initiator

```
8  boost::asio::io_service io_service;
9  udp::socket socket_(io_service, udp::
       endpoint(udp::v4(), port_));
10 socket_.async_receive(boost::asio::
       buffer(buffer_),boost::bind(&
       udp_server::handle_receive,this,
       boost::asio::placeholders::
       bytes_transferred));
11 for (int i=0; i < 24; i++)
12 boost::thread t(boost::bind(&boost::asio
       ::io_service::run,&io_service));
```

Sample program

Proactor design pattern: Completion Handler

```
13  handle_receive(const size_t&
        bytesReceived) {
14      char* data = buffer_;
15      buffer_ = malloc(9000);
16      // Let enother thread read next pack
17      socket_.async_receive(boost:::...
18
19      DO SOME WORK WITH data HERE!!!
20  }
```