Client supported environment

Programming languages

The protocol is open sourced and a client library can be implemented by using any language. Currently we support:

- Java 12 for desktop and Android applications
- TypeScript for web-based, node.js applications (that's the native TypeScript - not the emscripten-driven web-assembly port from C++ library)
- C++ 17 for Objective-C, Swift, Desktop, mobile applications and IoT devices
- ~~C is a wrapper over C++ API~~

C++

OSs (expandable list)

- Macos / iOS / iPadOS
- Linux
- Unix (Solaris, FreeBSD)
- Windows, mingw

IDEs (expandable list)

- Arduino Studio
- VSCode
- XCode
- MSVC
- ~~Keil~~
- ~~StmCubeMX~~
- PlatformIO

Command line tools (expandable list)

- ESP IDF
- gcc / clang / msvc CMake
- ~~Google Blaze~~

CPUs (expandable list)

- Intel x86, x86_64
- Arm
- ESP32

Minimum system requirements:

- Code binary size - 300Kb (expandable list)
    - with CRT and STL code

    - libhydrogen
    - TCP only
    - Fire-and-forget QoS level only
    - send message only
    - get destination cloud request is supported
    - Excluded modules:
        - ping
        - registration module
        - cloud DNS resolution
        - data compression
        - telemetry
        - latency statistics
- RAM usage - 30 Kb
    - the same configuration as for code binary size
- ROM usage - 200 bytes
    - pre-registered client
    - one server in the cloud
    - messaging to one destination client
    - only one server in the destination client cloud

Client libraries

Aethernet client is implemented natively in some languages with different sets of API functions supported.

Java

Java is a native language in which Aethernet server and database service are implemented. Java clients can be used for an application server due to the full set of functions implemented. It is not designed to work properly with a good Internet connection.

TypeScript

We implemented the client natively in TypeScript to simplify maintenance of the library on the user side. It also supports all functions and can be used as an application server with NodeJs.

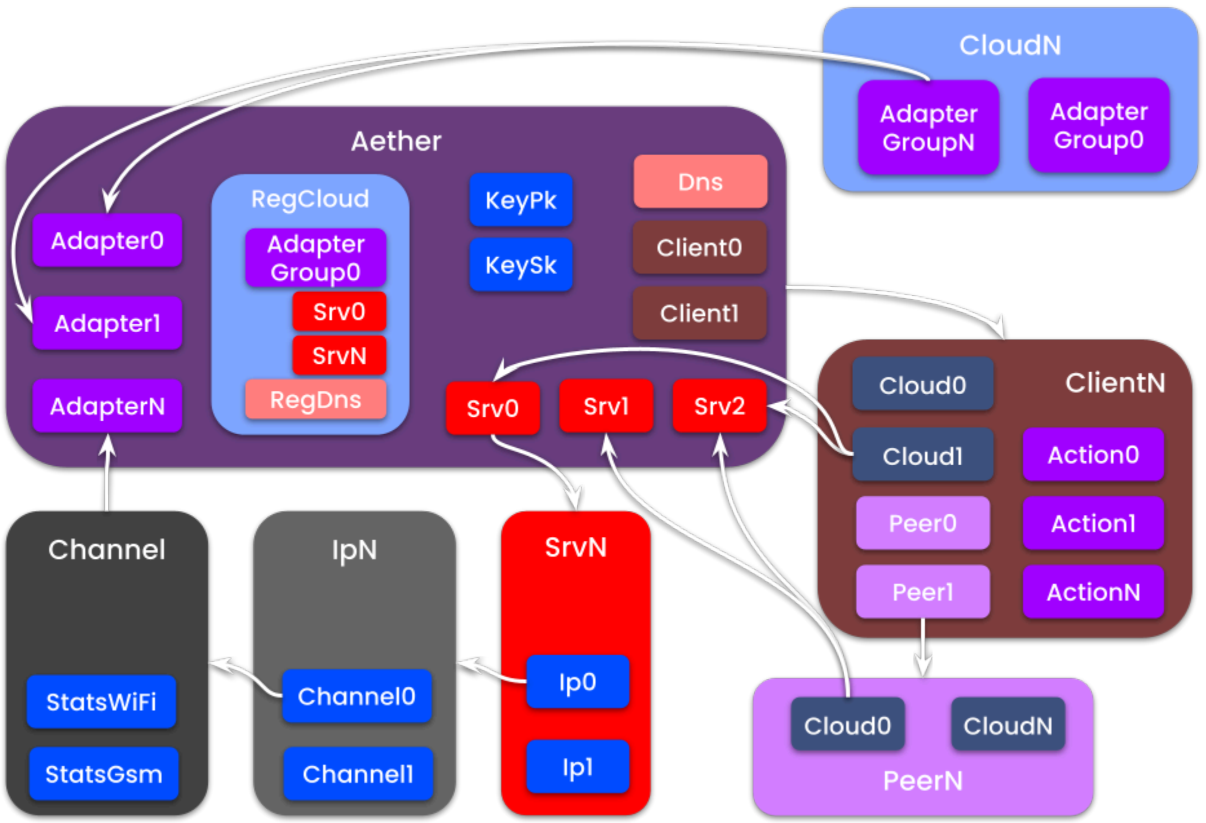
C++

The most advanced library that supports only a limited set of functions but with the stress of minimizing message delivery latency, remote updates, maximizing connectivity and portability. The library is not universal but highly customizable.

The client’s state

Aether API contains persistent state that is serialized / deserialized between API instantiations / releases. The state contains all necessary information about the Æthernet cloud configuration, clients sequence numbers, metrics etc. The state greatly improves connectivity and message delivery latencies.

State’s block diagram (C++ client library):



Each block in the diagram represents an object with a state that is serialized/deserialized in persistent storage between the Aether API runs. Exceptions are Actions, Adapters and AdapterGroups – these objects are released at the Aether shutting down and initialized at the API creation and during the API lifespan. The state of the instance of Aether API changes with new servers, clouds, statistics, clients, messages, requests and peers.

Aether C++ client architecture uses Object as a serializable primitive. All blocks on the diagram are instances of Obj. All references to objects are also serialized. It is possible to reconfigure the state, add / change / remove objects without recompiling the application by just changing the state binary files.
A newer version of the client library can contain only additional binary files. Forward and backward compatibility is automatically supported. A user can upgrade the application with the

newly compiled version that enhances some objects (say, adding new parameters in statistics). The new version loads the previous version state and serializes with additional data. Later, the user can revert-back the updated version to previous versions - the state is deserialized correctly. The user can upgrade the version again correctly.
Examples:

- if a server public key is changed then just a binary file can be supplied with an upgrade
- a new server's descriptor can be added by just binary file

Distillation mode

A default state is stored with a special distillation built by the developer of the application. This build contains a construction of all references to objects with all necessary data processing and verification. A release build strips out all the construction code that minimizes code size and possible bugs.

Partially loaded graph

To minimize RAM usage only needed objects are loaded. For example, a registration cloud with all server's descriptors are needed only at the client's registration action. It loads for the registration (all references to loaded objects are automatically restored) and unloads right after that.

Object system (C++)

C++ client architecture is built on Object system that allows to reach some goals:

- extremely fast automatic serialization / deserialization runtime components
- removing constructors by using the serialized binary state instead:
    - improves the startup performance due to pre-processing
    - eliminates the construction code with minimizing the code binary footprint, memory requirements
    - moving construction errors from product into development distillation stage
- implementing forward and backward compatibility with minimum additional effort
- partially loaded subgraphs minimize memory usage
- updates with just serialized data with no requirement for executable updates
- structures objects executing with no requirement for priority queues

Getting started

Æether Object is a C++17 cross-platform framework for creating lightweight, highly structured, easily supportable, reliable applications.

The core concept is well-known from other frameworks: an application is represented as a graph, and each node is a serializable class instance.

Hello, world!
In the example, an application graph is constructed with the application objects A and B and then serialized into the user-provided storage, for example, into files.

Then the application graph with the whole hierarchy can be restored from that serialized state.

```
#include "aether/obj.h"

class A : public Obj {
public:
  AETHER_OBJ(A, Obj);
  template <typename T> void Serializator(T& s) { s & i_; }
  int i_;
};

class B : public Obj {
public:
  AETHER_OBJ(B, Obj);
  template <typename T> void Serializator(T& s) { s & objs_; }
  std::vector<A::ptr> objs_;
};

int main() {
  constexpr uint32_t b_predefined_uid = 123;
  {
    // Serialize the state into files.
    Domain d{file_saver};
    B::ptr b = domain.CreateObj(B::kClassId, b_predefined_uid);
    b.objs_.push_back(domain.CreateObj(A::kClassId));
    b.Serialize();
  }

  // Loading the state from files
  Domain d{ remover, loader, enumerator, saver};
  B::ptr b;
  b.SetId(b_predefined_uid);
  b.Load(&domain);
  // Loads automatically by accessing, for example b->objs_.size()
}
```

Class pointer casting

Any class of an application is inherited from the Obj class. AETHER_OBJ(derived, base) macro is used to declare all supporting internal functions. An Object is wrapped into the Ptr template class and the pointer type is declared as MyClass::ptr. Obj base class contains the references counter.

```
Domain d;
MyClass::ptr c1 = domain.CreateObj(MyClass::kClassId);
auto c2 = c1;  // Increments the reference counter
c1 = nullptr;  // Decrements reference counter
c2 = nullptr;  // The class instance is released
```

Inheritance

Each class inherited from the Obj class supports efficient dynamic pointer downcasting without using C++ RTTI.

```
class A : public Obj {
  AETHER_OBJ(A, Obj);
};
class B : public Obj {
  AETHER_OBJ(B, Obj);
};
Domain d;
Obj::ptr o = domain.CreateObj(B::kClassId);
A::ptr a = o;  // Can't cast to A* so the pointer remains nullptr
B::ptr b = o;  // Resolved to B*, Increments the reference count.
```

Serialization

Serialization of the application state is done with an input/output stream that saves an object data and references to other objects. A special **mstream** class is provided. A user-side call-backs implement saving/loading the serialized data, for example as files, or database etc. Just a single method in a class must be implemented for serialization / deserialization. The method is also internally used for:

- building initial binary state and making pre-fabs in Distillation mode
- extracting subgraphs
- searching cycles in the graph
- building the object’s execution priorities
- serialization / deserialization object’s state
- versioning
- cloning objects

```
template <typename T> void Serializator(T& s) {
  s & my_data & my_pointer_to_other_objects;
}
```

Class state serialization

To avoid possible mismatches when class members are serialized and deserialized a unified bidirectional method is used:

```cpp
class A : public Obj {
 public:
  AETHER_OBJ(A, Obj);
  int i_;
  std::vector<std::string> s_;
  template <typename T> void Serializator(T& s) {
    s & i_ & s_;
  }
};
```

Serializator method is instantiated with an in/out stream and '<<', '>>' operators are replaced with a single '&'operator. It is possible to determine the type of the stream at compile time for creating some platform-specific resources:

```cpp
if constexpr (std::is_base_of<ae::istream, T>::value) { ... }
```

Class pointer serialization

A pointer to another object can also be serialized/deserialized in the same way like any built-in or STL type:

```cpp
class A : public Obj {
 public:
  AETHER_OBJ(A, Obj);
  int i_;
};

class B : public Obj {
 public:
  AETHER_OBJ(B, Obj);
  Obj::ptr o_;  // A reference to the A* class but casted to Obj*
  std::map<int, A::ptr> a_;
  template <typename T> void Serializator(T& s) {
    s & o_ & a_;
  }
  void Update() {
    A::ptr(o_)->i_++;  // Valid
    a_[0]->i_++;  // Valid
  }
};
```

If multiple pointers are referencing a single class instance then after deserialization a single class is constructed and then referenced multiple times. Cyclic references are also supported. Each class is registered with the factory function and the unique ClassId. Each class instance = object contains a unique ObjId. Both these values are used for reconstructing the original graph on deserialization.

Versioning

Versioning is implemented by inheritance chain and supports:

- an old serialized object's state can be loaded by the newer binary. Default initialization of the newly added values is performed from the default binary state and not from the constructor.
- An old binary can load a newer serialized state by rejecting the unused values. Another useful application of the versioned serialization is the upgrading application to newer versions (with ability to roll-back).

Example: V1 class serializes integer value. When the instance of the class is serialized through the pointer then the ClassId and ObjId are both serialized. Then the integer is stored.

```
class V1 : public Obj {
 public:
  AETHER_OBJ(V1, Obj);
  int i_;
  template <typename T> void Serializator(T& s) {s & i_; }
};
```

For a newer application version the V1 class is extended by inheritance:

```
class V2 : public V1 {
 public:
  AETHER_OBJ(V2, V1);  // Note: the base class is specified here
  float f_ = 3.14f;
  // Important: the method serializes only V2 data, V1 data is
  // already serialized in V1 class.
  template <typename T> void Serializator(T& s) {s & f_; }
};
```

When the class is serialized through the pointer then V1::ClassId is stored instead of V2::ClassId. V2 is the last class in the inheritance chain so it will be created with the CreateObjByClassId function that creates the last class in the chain. Then a separate blob of data will be stored with the V2's data - floating point number. If an older binary loads the serialized state then V1 class is created and the V2 data is ignored. If a newer binary loads the old data then V1::ClassId is loaded and V2 class is created but only V1 data is deserialized. V2 deserializes the data from the newer version default binary data.

Upgrading applications is easily implemented by replacing / adding serialization data for a particular class. All substates of all classes in the inheritance chain are stored individually.

Managing versions

Any application that includes the Æthernet C++ client should have a special build target for distilling the default state into binaries. The distilling process is driven by the AE_DISTILLATION macro. In this mode the internal client's state is maintained. The default state for the most recent version of the client library is stored in the folder "0".

A newer version of the library can contain code changes and/or state changes. If the code is changed then the developer application must be re-built, which can be painful for mobile applications due to triggering the approval process.
In contrast, if only the binary state is changed then the incremental or cumulative update can be performed with just storing new state into the state folder on the client device.

When the client library source code is updated from the github then the distillation should be performed if the library version has been changed. "1" is the folder name that contains the new full state. The update itself is a difference between "1" and "0". It's up to the developer to jump over several library updates. Later, if a developer decides to use a previous version of the library for some reason, then it is not possible to roll it back to the intermediate version.

Hibernating, Waking-up

An application is represented as a graph and some subgraphs can be loaded and some can be off-loaded at any moment. For example, an application can open a document while other documents remain off-loaded. Obj::ptr represents a shared pointer with reference-counting and the object can be loaded or not. When the pointer is serialized and then deserialized then the loaded/unloaded state is preserved. An object holding the unloaded reference to another object can load the object at any given time:

```
Doc::ptr doc_;
void SomeMethod() {
  // Not necessary to be called because any reference to the
  // object performs loading
  doc_.Load();
  // Some user-defined state change.
  doc_->AddString("example of method call");
  doc_->Serialize();
  doc_->Unload();
}
```

The doc_ is loaded from the saved state. Then the state of the object is changed and then the object is serialized with the new state. The object is unloaded then but it can remain loaded. The loaded/unloaded object must be referenced only by a single pointer. If an unloaded object's pointer is copied then the copy is nullptr.

User-defined callbacks are passed into the Domain class to allow objects state storing, loading and enumerating:

```
using StoreFacility = std::function<void(const ObjId& obj_id, uint32_t
class_id, ObjStorage storage, const ae::mostream& os)>;
using EnumerateFacility = std::function<std::vector<uint32_t>(const ObjId&
obj_id, ObjStorage storage)>;
using LoadFacility = std::function<void(const ObjId& obj_id, uint32_t
class_id, ObjStorage storage, ae::mistream& is)>;
```

In the example application a file storage is used:

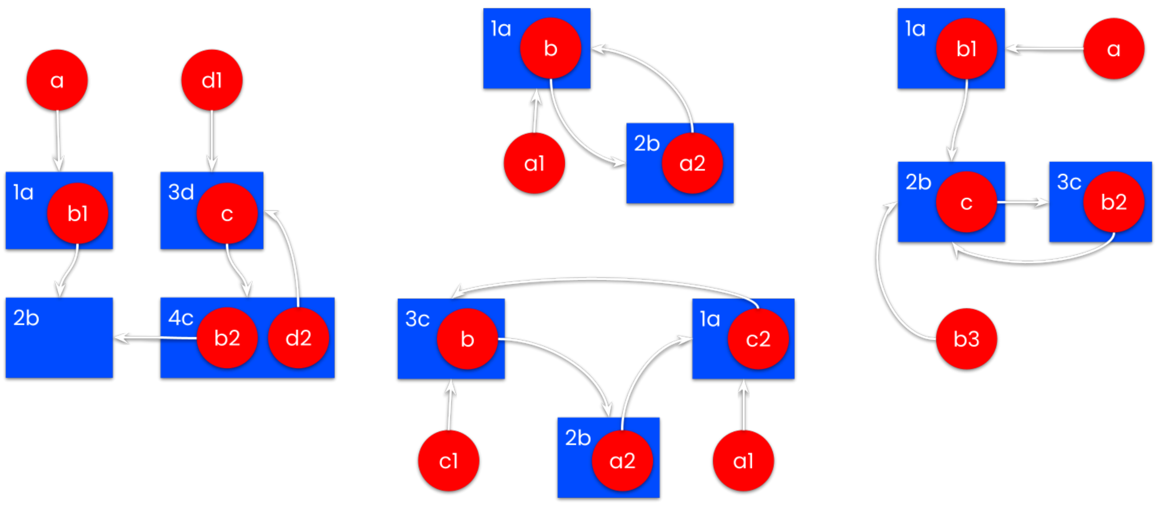
- each object is serialized into the separated directory
- InstanceId is the name of the directory
- a separate file with the name of class_id for each class in the inheritance chain is use for storing the data
- the whole graph of the application is linearized into plain structure where all objects are placed on top level

Multiple references
When an object's pointer is deserialized the object is being searched with the unique ObjectId if the object is already loaded by another upper-level node. If it is loaded then it's just referenced. If the object is referenced multiple times and the pointer is unloaded then the object remains alive.

Cyclic references
For a particular object pointer that references other objects and is to be unloaded only objects referenced within the subgraph are unloaded. That also includes cyclic references:

```
class A { B::ptr b_; };
class B { A::ptr a_; };
A::ptr a;
a.Unload();  // B and A referenced with the subgraph only
```

Some examples of graphs that are correctly handled by the object system. For example, in the first picture if **d1** is released then **1a** and **2b** objects are still kept.



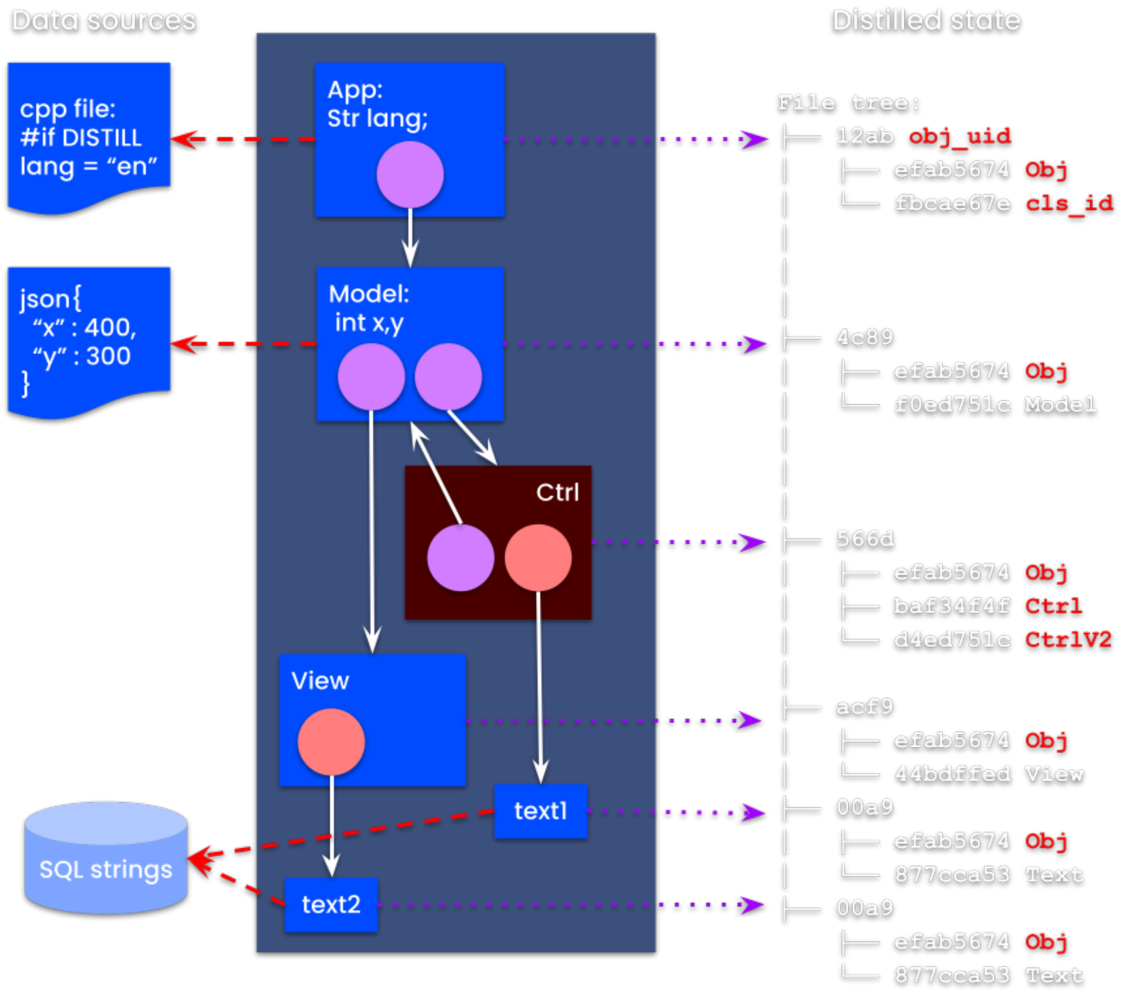
Example graph

The example below shows the application that starts from the root node and contains string and a pointer to the Model both initialized on distillation mode from the code.
The distilled state is generated with the FileSystem adapter.
12ab - the folder name that means the unique object ID.
The folder contains files with names:

- **efab5674** - Obj::kClassId, the file itself contains Obj data members
- **fbcae67e** - App::kClassId, the file contains the string and the reference to the **Model**

"**Ctrl**" object has 2 versions. The data for different versions is stored separately into different files.

The distillation process can take data from different sources as shown in the picture below: in-code initialization, json file and database.



```
File tree:
├── 12ab obj_uid
│   ├── efab5674 Obj
│   └── fbcae67e cls_id
│
│
│
├── 4c89
│   ├── efab5674 Obj
│   └── f0ed751c Model
│
│
│
├── 566d
│   ├── efab5674 Obj
│   ├── baf34f4f Ctrl
│   └── d4ed751c CtrlV2
│
├── acf9
│   ├── efab5674 Obj
│   └── 44bdffed View
├── 00a9
│   ├── efab5674 Obj
│   └── 877cca53 Text
├── 00a9
    ├── efab5674 Obj
    └── 877cca53 Text
```

Telemetry

Aethernet C++ client contains a telemetry module that:

- gather run-time device information
- counts functions invocations
- measures duration between control points
- collects logs

The module can be stripped-of completely to reduce binary size or can be included in full scale. Aethernet can request telemetry that later can be retrieved then by an application server. It is strongly recommended to include the telemetry module except logging.

```
AE_TELE_INFO(“Label”, “Console text with param {}”, float{1.0f});
```

“Label” is a string that is translated into a single byte for the most frequently used values. Next, a text that appears in console or debug output and completely stripped from executable if not used. A list of parameters used only when logs are enabled. Duration measurement is also optional. Every AE_TELE is translated into the invocation counting, duration measurement and logging - all are optional. Statistics are collected for each label such as: invocations count, min/avg/max duration of living.

Telemetry is the compile-time library that contains no string identifiers in the compiled executable. It uses just a single byte per label for logs (if log option is enabled). Durations are also compressed into a single byte allowing telemetry to be used even on constrained embedded systems.
Telemetry is used by Aethernet to greatly improve user experience with the client library in all aspects: bugs, execution time, memory consumption for RAM and for Flash.

The Telemetry module can be configured by including 4 levels. Adding a level requires to include all previous levels:

bytes:	Code	Ram	Flash
Environment			
Stats			
Logs		1 byte / line	0
Console			

Numbers

Aethernet C++ client is targeted for restricted devices with limited memory size, bandwidth and no floating point unit presented. For that reason special formats of numbers are implemented.

Packed

That's the integer format that serializes with packing of the most frequently used numbers into lower count of bytes. For example, all numbers less than 200 are fit into a single byte, less than 4000 fit into two bytes etc. The format is represented as a built-in type at run-time and compresses / decompresses the value during the serialization / deserialization step only. There are a lot of already existing libraries like in Google patent or UTF-encoding for the variable length encoding. Our library allows to choose ranges of numbers for a particular number of bytes flexibly that improves the packing ratio. The compression-decompression code is also very small.

An example of increased packing efficiency is when the size of payload of a datagram can't be greater than 1500 bytes. In a traditional variable size packing it takes one byte for values less than 128 but with our implementation values less than 251 are represented by a single byte.

```
Packed<uint32_t, 2, 16, 256, 5, 16, 256>
```

- uint32_t - the type is represented as uint32_t at run-time
- 2 is the minimum number of bytes to be encoded into
- 16 values are reserved from the first two bytes into next bytes. Maximum value encoded by a single byte is **240**
- 256 values are reserved from two bytes encoding to the rest of the values. It allows 1536 numbers to be represented by two bytes.

Fixed

A floating point unit is not always included in the microcontroller or the performance of the unit is not sufficient. Also the FPU can support only 32 bit floating point and the dynamic range is large with the price of decreased resolution.
The fixed point module allows to use these numbers with the same semantics of floating point numbers without thinking about the range and possible overflow.
There are a lot of 3d party fixed point libraries [1, 2, 3] but our library allows to not define the number of bits allocated for the integer part directly, instead the number of bits for the integer part is calculated at compile-time from the range defined as floating point.

```
AE_FIXED(uint8_t, 123.5) f(3.14f);
```

In addition to that the value can be initialized right from the floating point representation at compile time.

```
AE_FIXED(uint8_t, 10.0) f1(3.14f);
AE_FIXED(uint8_t, 10.0) f2(9.0f);
auto f3 = f1 + f2;
```

The library can deduce the type of the result of arithmetic operation to avoid overflow. In the example above the result type range is [0..20].

Another great feature of the library is that it can support out-of-range position of the fixed point:

```
AE_FIXED(uint8_t, 60000.0) f(3000.0f);
AE_FIXED(uint8_t, 0.0001) f(0.00001f);
```

Other libraries allow setting the position of the point within the range [0..8].

Exponent

Exponential number is a compact representation when the high range should be represented and relative precision should be used instead of absolute precision. For example, a function execution duration can take from several microseconds to several seconds. When it takes seconds then it is not practical to have microseconds precision.
The exponential number defined at compile time specifies the run-time type to be used and the exponent within the base that is used for serialization / deserialization.

```
using E = AE_EXPONENT(uint8_t, 0.001, 60.0);
```

Combined types

Any combination of the types defined above can be used.

The exponential number where small numbers are encoded into a single byte:

```
using P = Packed<uint16_t, 1, 16, 256, 5>;
using E = AE_EXPONENT(P, 1.0, 600000.0);
```

The same but for fixed point numbers:

```
using F = AE_FIXED(uint8_t, 0.001);
using E = AE_EXPONENT(F, 0.00000001, 0.001);
```

Combining packed integer, fixed point and exponential numbers:

```
using P = Packed<uint16_t, 1, 16, 256, 5>;
using F = AE_FIXED(P, 0.001);
using E = AE_EXPONENT(F, 0.00000001, 0.001);
```

~~Wrappers for C++ library~~

~~C~~

~~Designed to simplify Aethernet library integration into C projects primarily for microcontrollers.~~

~~ObjectiveC~~

~~Bridge packed as XCFramework for both x86_64 and Arm for macos / iOS / iPadOs / AppleTvOs~~

~~Swift bridge for ObjectiveC~~

~~Bridge packed as XCFramework for both x86_64 and Arm for macos / iOS / iPadOs / AppleTvOs~~

~~Java NDK~~

~~The difference between native Java library and theNDK wrapper is that the NDK version is targeted for client devices but not for the application server. This version provides remote updates, best customization etc. - the same as C++ version.~~

Command line interface

Command-line utility is useful for managing the infrastructure. It allows to create all necessary nodes, to gather statistics, to find anomalies and to perform any operations on clients, including managing quotas. Aether CLI "aether-cli" built on Java client.

Hierarchy

Æthernet is a hierarchical structure of nodes starts from the root node “Aether”:

- **users** - all clients associated with the web-site accounts
- **anonymous** - self-provisioned clients. A client can register themself with a price of proof-of-work
- **examples**
    - **chat** - the chat application with just a single chat room
    - **echo** - any client is able to receive messages from any other client

Syntax

```
aether-cli command file_name
```

An Æthernet client once created has a state that is serialized into the file. Every command requires the actual state to be used.

Creating root client

Provisioning a client

A client always has a parent client to be specified at the creation time.

```
aether-cli create file_name parent_uid
```

parent_uid is placed in hex or text as “anonymous”, “test”

Retrieving data from the client state

```
aether-cli get file_name type
```

where **type** can be:

- uid
- master-key
- cloud servers

Set/Get

```
aether-cli set/get file_name type
```

type as follows:

Registration

registration-subtype

rate

In registrations per second that the Æthernet will target by increasing/decreasing the work factor for the application. Example:

```
aether-cli set client_state.bin registration-rate 100
```

allowmsg

Allows a newly created client to receive messages from the specified client. A particular client can be specified and/or a flag indicating that messages can be received from the child clients:

```
aether-cli set client.bin registration-allowmsg 0123456789abcdef subtree
```

Multiple allowances are permitted.