Æthernet documentation

Документация

- aether - concept
    - software architecture - back-end
        - clients, m2m communication and online presence
        - application
            - no message storage, database - just messaging and management
        - aether is a glue that connects clients and the application server and takes many infrastructure tasks:
            - client authentication
            - DoS protection of aether, applications and clients
    - management
        - hierarchical clients relations
            - transfer clients
        - quotas
        - permissions
    - cloud
        - client-server
        - client makes requests to one or more closest servers
        - replication by sender on multiple destinations
    - Protocol
        - no round-trips - stateless
        - pull-only
        - redundancy with multiple servers
        - message redundancy
    - pricing as pay-as-you-go
        - zero-balance
    - supported platforms
        - Java - android
        - Js - web, web apps
        - C++
            - Desktop: win, mac, lin
            - Mobile: apple
            - IoT: arduino, stm, esp
- Client lifecycle management
    - registration / new client
    - sending messages
    - restricting / deleting
- app server
    - managing clients

- Connectivity
    - pull-only
    - no round-trips
    - multiple protocols
    - multiple end-points simultaneously
    - dns resolving
- Crypto
    - sodium…
        - padding / oracle / reply
    - server-side key management, key pinning
- DoS
    - amplification attack
    - registration vs working cloud
    - proof of work against Sybil
    - TCP / UDP

Getting started

C++ client library

The C++ client library is a highly customizable cross-platform solution that provides an API for a client application to interact with Aethernet. Here are minimal necessary steps to follow for the message sending and receiving. (The namespace "ae" is omitted).

Aether is an instance of the API. An application can create multiple instances of Aether (for debug purposes, for example). A domain is a container of the API instance objects and must be unique for each instance of the API.

```
Domain domain;
Aether::ptr aether = FileSystemAdapter(domain);
```

Adapter is a runtime-specific class that implements network interaction, for example, with Berklay sockets. An adapter must be constructed from the adapter's prefab:

```
Adapter::ptr adapter{
    aether->adapter_prefabs_.front().Copy(*aether->domain_) };
```

C++ client passes each request to Aethernet through all **adapter groups** assigned to the client's cloud simultaneously. A registration and a working cloud are separated. An Adapter group defines a policy of how the request is passed through the groups' adapters.

```
AdapterGroup group1{adapter1, adapter2};
AdapterGroup group2{adapter1, adapter3};
aether->registration_cloud_->adapter_groups_ = {group1, group2};
```

An application may create a **Client** through the Aether API. Multiple simultaneous clients are supported. A client is represented by an unique (across the whole Aethernet) identifier.

```
Client client = aether->NewClient();
```

Any interaction with Aethernet is done with Actions. The action performs all necessary requests to Aethernet. The action provides the progress of an operation, can be gracefully canceled or terminated. All actions are released on application shutting down.

A client must be registered first, to get the uid:

```
Registration::Descriptor desc{parent_uid};
Registration action = client->TakeAction(desc);
while (!action->Done()) { sleep(1); }
```

An action progress can be asked or a call-back can be used for the progress notification.

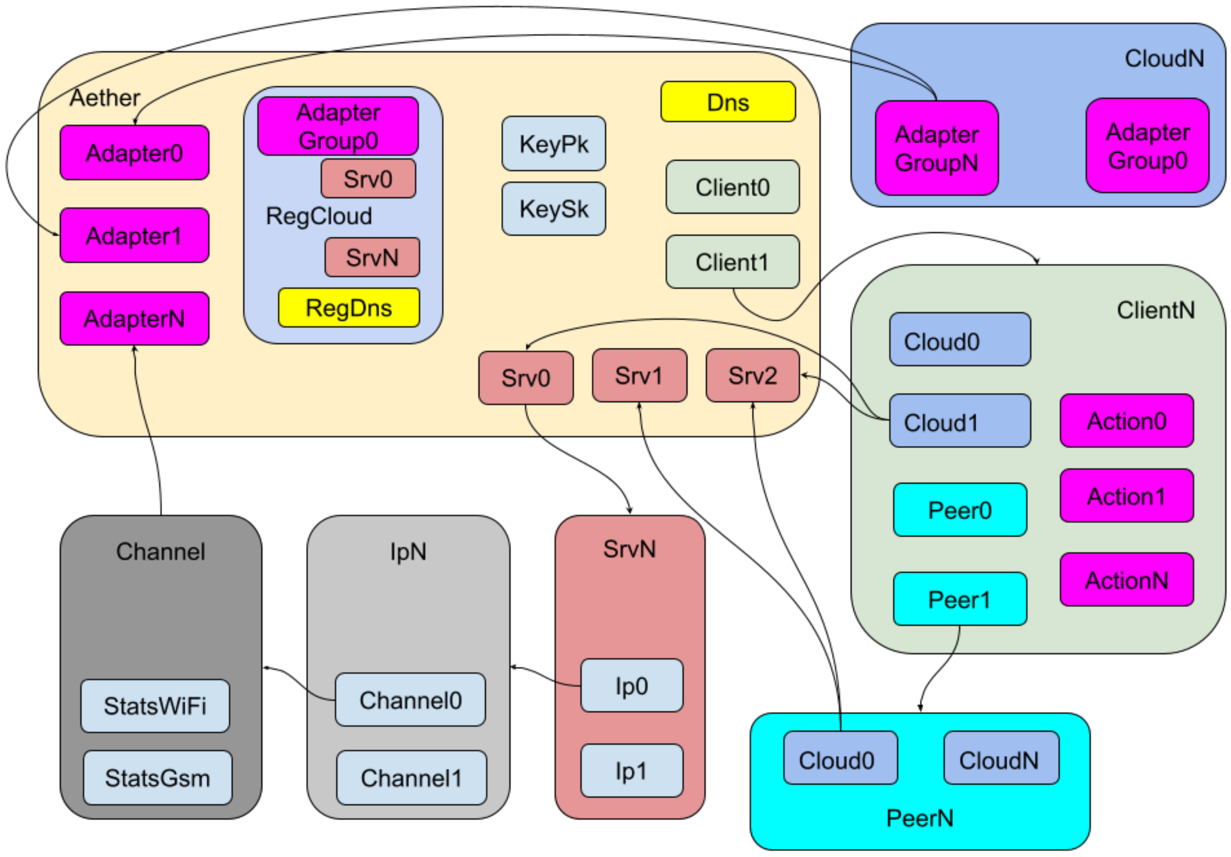
A client can send a message to another client with the action:

```
SendMessage::Descriptor desc{addressee_uid, data, data_size};
SendMessage action = client->TakeAction(desc);
```

A client can pull messages:

```
Pull::Descriptor desc{pull_interval, message_received_callback};
Pull action = client->TakeAction(desc);
```

Block diagram:



Each block in the diagram represents an object with a state that is serialized/deserialized in persistent storage between the application's runs. An exception 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.

The state

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

Instrumentation mode

A default state is stored with a special instrumentation 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.

See section "Object system" for more information.

The life cycle of an Action

A newly taken action is stored into the client's action list. The action executes periodically. The action can perform a request to Aethernet by sending the request to the ae::Cloud. The action is notified when a reply/request from the server is received. A request may contain several methods. The action inspects methods and performs some operations:

- changes aether / client / cloud etc. states, for example, adding new servers into the cloud
- releases themself or other actions
- creates other actions

The life cycle of a request

The cloud sends a request through all adapter groups simultaneously. A single adapter from a group is chosen based on the adapter's policy.

The end-point of Aethernet is identified as:

- server_id - a physical server
- IP address. Multiple IP addresses can exists for a single server:
    - IPv4 and IPv6
    - multiple network adapters can be installed
- protocol and multiple port numbers which supports the protocol
    - TCP
    - UDP
    - HTTP
    - HTTPS
    - WebSocket

- A proxy can be used for all connections

A channel is a physical link to the end-point that can be represented by a socket, for example. Multiple channels to a single end-point can exist within a single or multiple adapter.
A cloud serializes the request into a binary blob and examines ways to send the blob to end-points. The cloud calculates a delay of delivering the blob by taking into account:

- previous measurement done for the end-point
    - socket creation and connection time (for TCP)
    - delivery latency for a specific amount of data to be sent (bandwidth)
    - the implementation specific network adapter feature,an adapter can support
        - non-blocking mode
        - epoll-like notification mechanism
        - DMA into the user space directly without involving OS kernel
- currently existing connection (if any). No connecting delay is involved
- server response time - it also includes processing time that is a bit higher for HTTPS, for example
- currently existing channels
    - the output buffer can contain some data that is waiting to be sent - additional delay
    - the data in the output buffer that can be dropped, if can (for methods with higher priority)

The cloud, once the channel is chosen, sends the data blob piece-by-piece with as small portions as possible to have more control over the delay and priorities of other methods. The progress of the request can be retrieved with the expectation of when the request is going to be completed.

A channel is a stream-oriented pipe that contains methods - a blob of data that is interpreted specifically based on the method's identifier.

Methods

Method is a core concept of Aethernet protocol. Servers and clients both execute the logic based on methods. A method is defined with the method identifier (one byte for the most frequent methods and two bytes for the rest). A method can include other methods on which the method performs some data transformations like encryption. An example of zero-level message:

- Encrypted with libsodium secret key
- Signed by libhydrogen
- Ephemeral uid and encrypted with session key with libsodium

Zero level means that the first byte that is coming out of the channel is interpreted as an identifier of the zero-level namespace of methods.
For example, the encryption method decrypts the data blob it contains and passes the blob into the method parser into the next level of the methods' namespace:

- send a message
- compressed data

- pull messages

Compressed data method can process data and pass into the compression namespace:

- zlib compressed
- lzma
- etc.

The Lzma compressed method uncompresses the data and passes the blob to parse methods back into the Level-2 namespace (send message, compress data etc.).

The logic allows to minimize usage of method IDs which is very important due to the protocol versioning and cryptographic safety.

Safety

Level-0 namespace of methods identifiers defines cryptographic methods so the first byte (method id) can't define any other methods, preventing usage of non-encrypted methods.

Versioning

Aethernet protocol versioning is done via adding new methods only. All Aethernet servers support the latest version of the protocol. A client can be upgraded at any time and newly implemented methods can be used. Downgrading a client also works so just a subset of supported methods is used.
A client can be configured excluding some functionality and methods support. The client telemetry can be sent to the server to allow the server to understand what methods are supported when the server makes push requests.
If no telemetry is sent then the server assumes that the client supports just the base methods.

Merging

A request contains a tree of methods, an example:

1. Encryption with libsodium: nonce
    a. pull message
    b. compression
        i. zlib
            1. send message 1
            2. send message 2

Zlib serializes underlying methods into the blob and then compresses the data.
Compression just passes the data it contains.
Encryption serializes the pull-method and takes the compression's blob, then encrypts the resulting blob.
If the cloud has several requests in the queue then the cloud can reassemble methods by merging some of them under the same encryption method.

Object system (C++)

Getting started

Æether Object is a C++17 cross-platform architecture header-only 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);
}
```

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 used for:

- building initial binary state and making pre-fabs
- 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:

```
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:

```
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 type:

```
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);
  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 remains in default value.

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

If the versioning is not intended then AETHER_OBJ(ClassName) should contain only a single class in the list. Also all members of the parent class must be serialized.

Hibernate, Wake-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() {
  doc_.Load();
  doc_->AddString("example of method call");  // Some user-defined state
change.
  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:

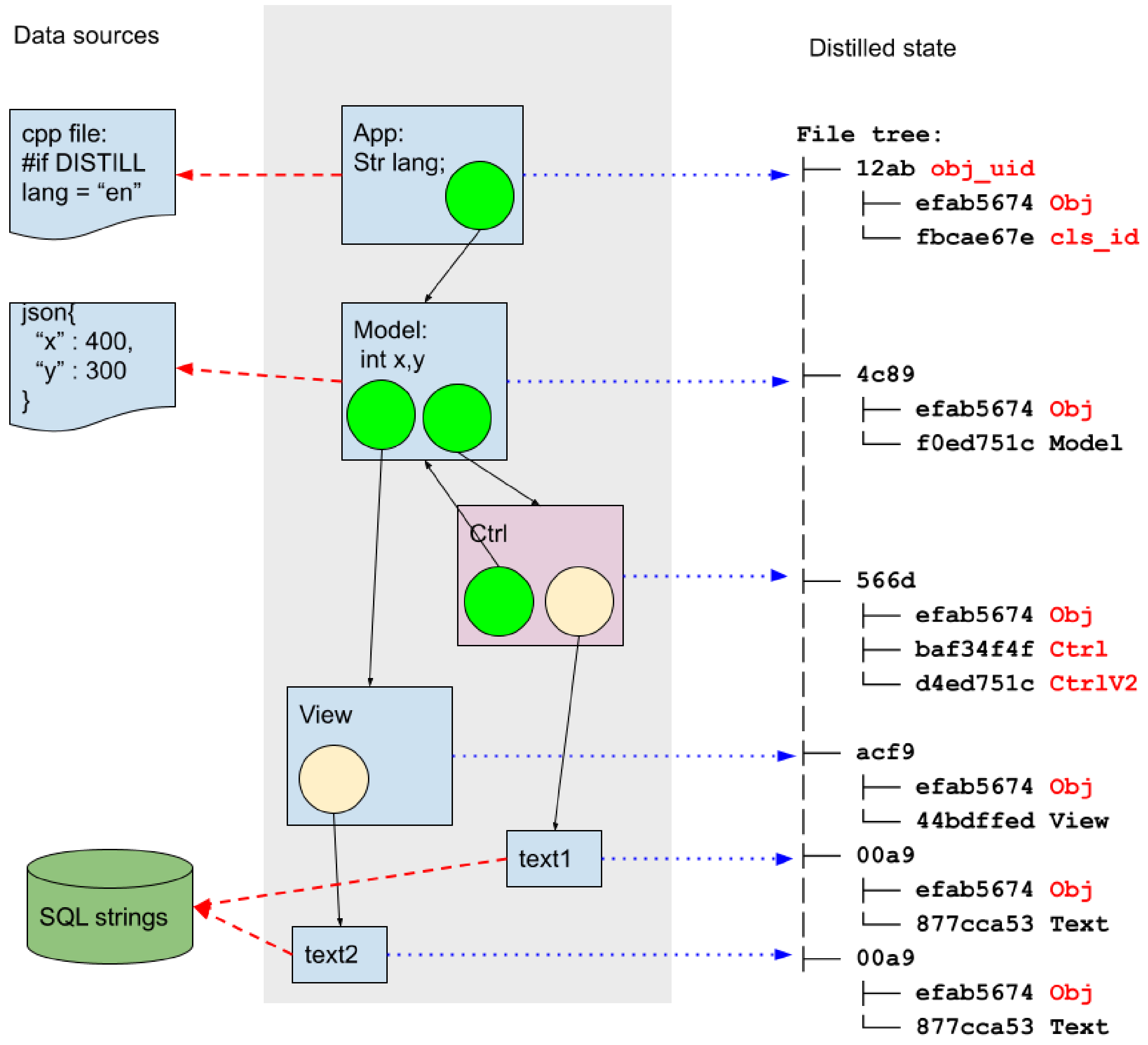
each object is serialized into the separated directory
InstanceId is the name of the directory
a separate file with the name of class_id for each class in the inheritance chain is use for storing the data
the whole graph of the application is linearized into plain structure where all objects are placed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

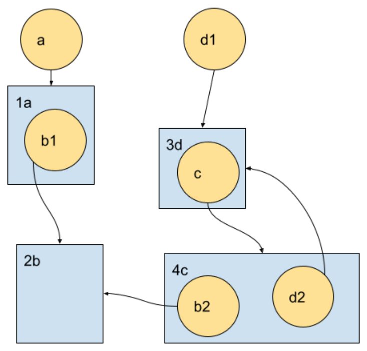
class B { A::ptr a_; };
A::ptr a;
a.Unload(); // B and A referenced with the subgraph only
Cyclic reference example

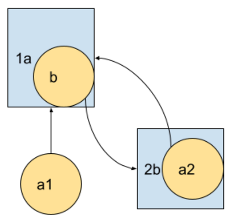
If pointer d is unloaded then objects D and C are also unloaded. Objects A and B remain alive.

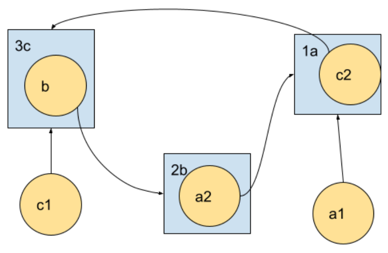


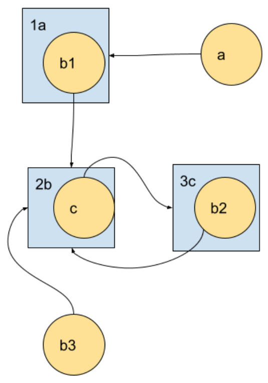
```
File tree:
├── 12ab obj_uid
│    ├── efab5674 Obj
│    └── fbcae67e cls_id
│
├── 4c89
│    ├── efab5674 Obj
│    └── f0ed751c Model
│
├── 566d
│    ├── efab5674 Obj
│    ├── baf34f4f Ctrl
│    └── d4ed751c CtrlV2
├── acf9
│    ├── efab5674 Obj
│    └── 44bdffed View
├── 00a9
│    ├── efab5674 Obj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

sdfsdf


a
d1
1a
b1
3d
c
2b
4c
b2
d2


1a
b
2b
a2
a1


3c
b
1a
c2
2b
a2
c1
a1


1a
b1
a
2b
c
3c
b2
b3

sdfsdf