

arataga

SObjectizer and RESTinio in action:
a real-world example

Foreword

What is 'arataga'

arataga is a fully working prototype of performant socks5+http/1.1 proxy server targeted a case when thousands of entry points have to be opened.

A way to open source

arataga was in development in the summer of 2020 by stiffsteam for a customer who wanted to replace its own old proxy server.

For some reason, the customer abandoned the project.

But almost all functionality we wanted to have was already implemented, so we decided to open source *arataga*.

Why we open sourced aragata?

It's a shame to throw away a product that our hard work was put into.

Maybe the results of our work will be useful for somebody.

It's also a perfect showcase of how we write code using SObjectizer and RESTinio.

Where arataga lives?

<https://github.com/Stiffstream/arataga>

Requirements for *arataga* and design decisions

Projects requirements

- thousands of entry-points (unique pairs of IP-address+TCP-port).
At least 8K entry-points should have been supported at the beginning;
- tens of thousands of users can work with the proxy simultaneously (40K parallel connections is a normal workload);
- user connections/disconnections at the rate of 1,000+ new connections per second;
- bandwidth had to be limited for client connections, including limits for a particular domains;
- the configuration and lists of allowed users must be received via a special HTTP entry point.

Multithreading or multiprocessing

Tens of thousands of connections are expected. They hardly be handled in a single-thread application.

We didn't go in the multiprocessing application direction: too many IPC was required for controlling child processes and collecting the monitoring info.

We decided to build a multithreaded application with several worker threads that serve connections.

Multithreading model

The old proxy server used by the customer adopts the thread-per-connection model.

It doesn't work well when there are more than 30K connections.

So we chose something like the thread-per-core model:

- $(n\text{CPU}-2)$ threads allocated for serving connections. We call them `io_threads`;
- there are also several service threads for working with configuration and so on.

Admin HTTP-entry

One of the service threads is dedicated to handling the admin HTTP-entry point.

That HTTP-entry point is used for accepting a new config and/or new list of allowed users and their parameters.

That HTTP-entry point is also used for retrieving the current stats from the proxy server.

What's next?

The next part of this presentation tells about the usage of SObjectizer to simplify multithreading.

Then we'll talk about the usages of RESTinio, some of that could be somewhat surprising.

SObjectizer's related part

Actors instead of raw
multithreading

Raw multithreading is a pain

Dealing with raw multithreading in C++ is a straight way to various problems.

We prefer to use more high-level approaches like Actors or CSP.

That is why we developed and maintain SObjectizer that allows us to use Actors and CSP when we need it.

Actors were an obvious choice for aragata

SObjectizer supports Actors and CSP models but in the case of *arataga* the choice was clearly obvious.

We have to deal with a large number of independent objects with their own state and logic.

Those objects required async interaction between each other.

That's an ideal use-case for actors.

Types of actors in arataga

There are a few types of actors (agents) in *arataga*:

- startup_manager;
- config_manager;
- user_list_manager;
- authenticator;
- dns_resolver;
- acl_handler.

startup_manager agent

startup_manager is used only at the start up.

It creates config_manager and user_list_manager agents.

Then launches RESTinio-based server for admin HTTP-entry.

config_manager agent

Tries to load local copy of the config at the start up.

Handles updates for the config from admin HTTP-entry.

Creates instances of asio_one_thread dispatches, creates and binds authenticator and dns_resolver agents.

Creates and destroys acl_handler agents.

user_list_manager agent

Tries to load local copy of the user-list at the start up.

Handles updates for the user-list from admin HTTP-entry.

Distributes updated versions of user-list to make them available for authenticator agents.

authenticator agent

Services authentication and authorization requests for `acl_handler` agents.

There is a separate authenticator agent for every `io_thread`.

dns_resolver agent

Services DNS lookup requests for acl_handler agents.

There is a separate dns_resolver^{*} agent for every io_thread.

^{*}there is no such agent as dns_resolver since v.0.4, now it's a coop of several agents. We'll see that later

acl_handler agent

Serves a single entry-point:

- creates an incoming socket for specified IP-address and TCP-port;
- accepts new connections for that entry-point (stops accepting when there are too many parallel connections);
- handles all accepted connections and all outgoing connection to target hosts.

Worker threads

There is no manual work with threads in *aragata*.

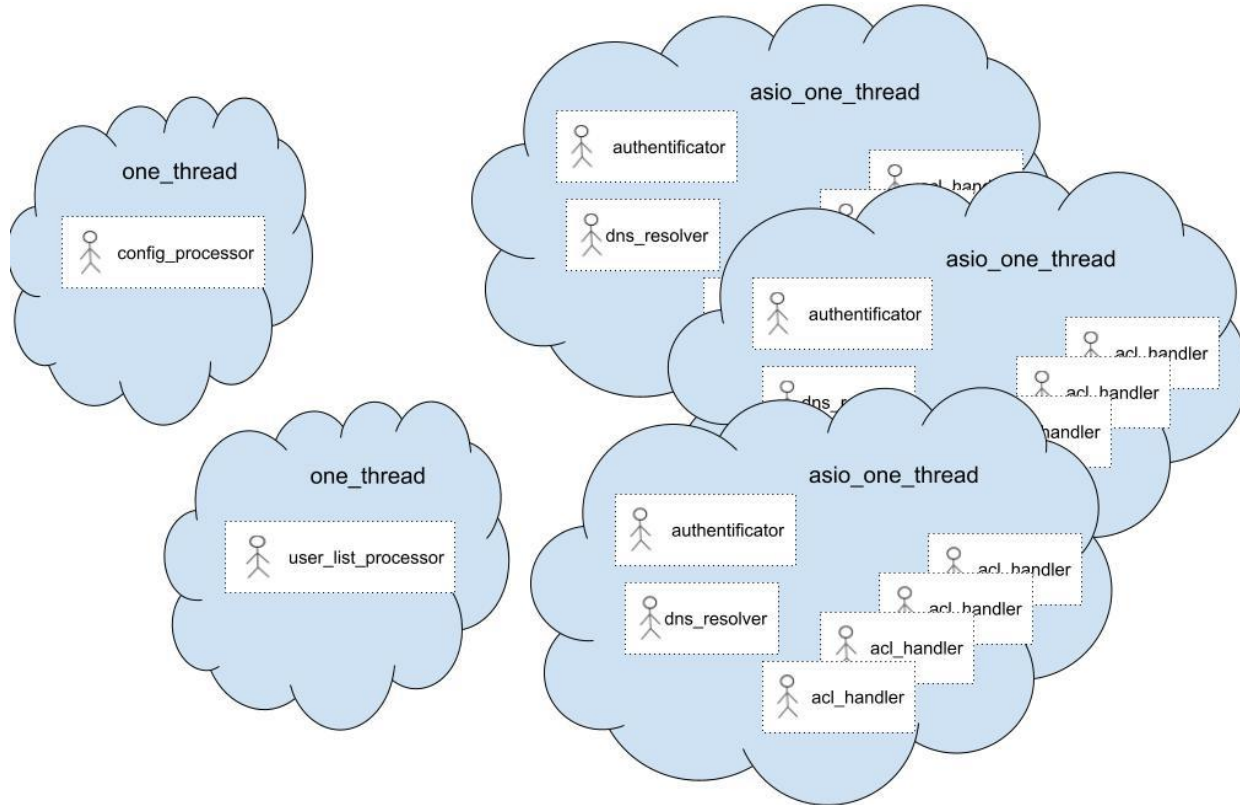
All worker threads required for *aragata*'s agents are automatically created and destroyed by SObjectizer. It's done by SObjectizer's *dispatchers*.

Dispatchers used in arataga

arataga uses two types of dispatchers:

- *one_thread* from SObjectizer's core. `config_manager` and `user_list_manager` are bound to instances of that dispatcher type;
- *asio_one_thread* from so5extra companion project. `authenticator`, `dns_resolver` and `acl_handler` agents are bound to instances of that dispatcher type.

Agents and dispatchers: the picture



What is asio_one_thread dispatcher?

This dispatcher holds an instance of `asio::io_context` object.

The dispatcher starts a new thread and calls `asio::io_context::run` on it.

All events of agents bound to that dispatcher are scheduled via `asio::post`.

That allows agents to use Asio's calls (like `async_accept`, `async_read`, `async_write`) directly in their event handlers.

This feature is critical for `dns_resolver` and `acl_handler` agents.

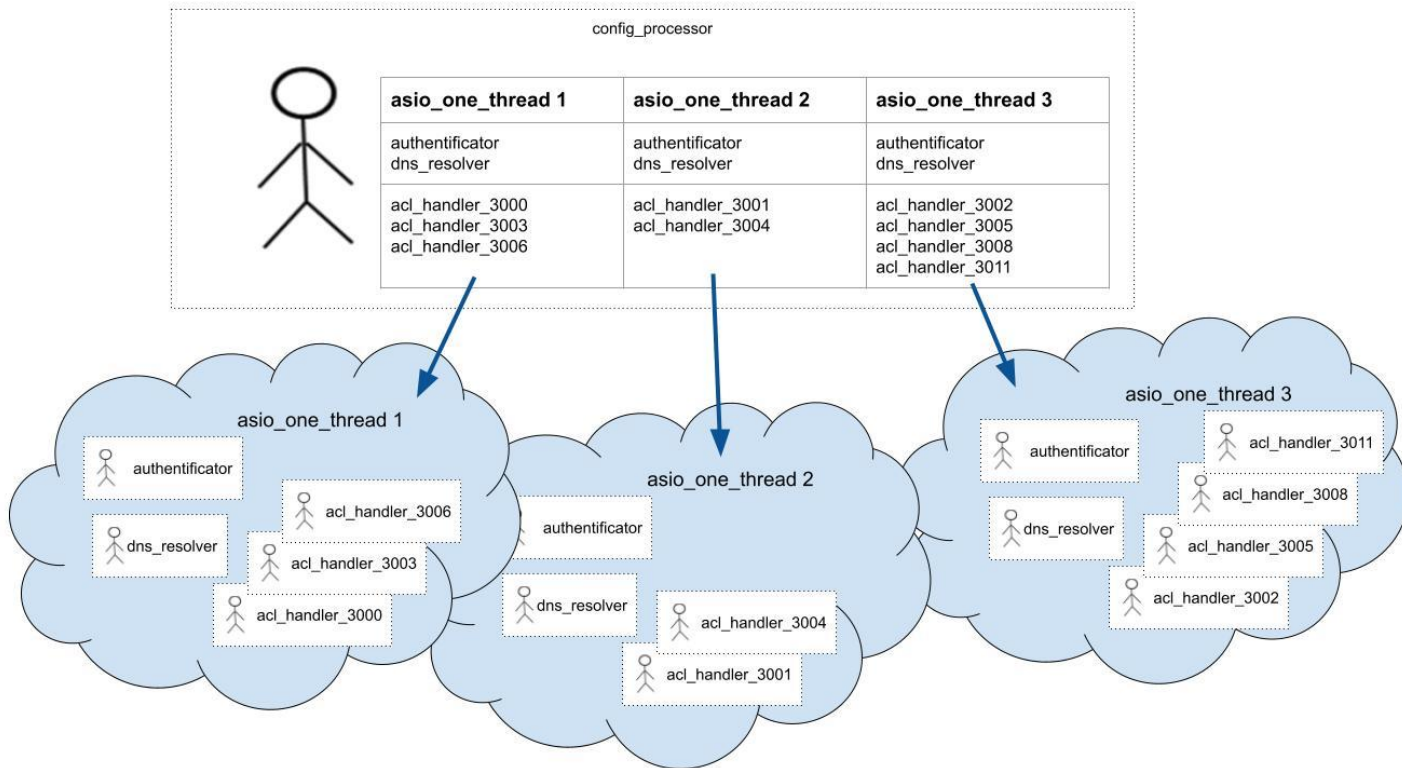
config_manager controls asio_one_thread disps

config_manager creates all necessary instances of asio_one_thread dispatchers at the start up.

config_manager holds references to them and uses them to bind new instances of acl_handler agents.

It can be seen as a table inside config_manager where a column describes one io_thread.

config_manager controls asio_one_thread disps

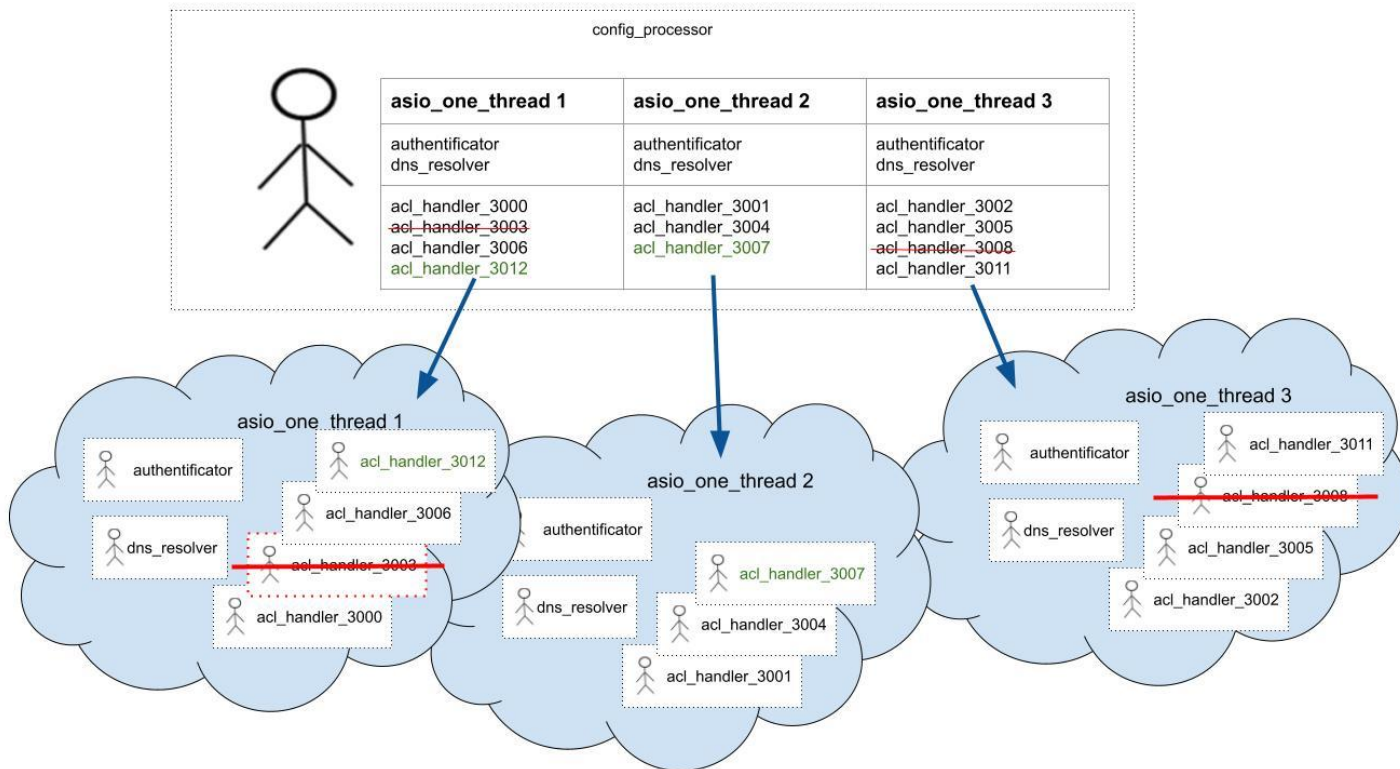


config_manager controls acl_handler agents

config_manager creates and destroys acl_handler agents when the config changes.

```
acl auto, port=3000, in_ip=127.0.0.1, out_ip=192.168.1.1
acl auto, port=3001, in_ip=127.0.0.1, out_ip=192.168.1.1
acl auto, port=3002, in_ip=127.0.0.1, out_ip=192.168.1.1
-acl auto, port=3003, in_ip=127.0.0.1, out_ip=192.168.1.1
acl auto, port=3004, in_ip=127.0.0.1, out_ip=192.168.1.1
acl auto, port=3005, in_ip=127.0.0.1, out_ip=192.168.1.1
acl auto, port=3006, in_ip=127.0.0.1, out_ip=192.168.1.1
-acl auto, port=3008, in_ip=127.0.0.1, out_ip=192.168.1.1
+acl auto, port=3007, in_ip=127.0.0.1, out_ip=192.168.1.1
acl auto, port=3011, in_ip=127.0.0.1, out_ip=192.168.1.1
+acl auto, port=3012, in_ip=127.0.0.1, out_ip=192.168.1.1
```

config_manager controls acl_handler agents



The role of acl_handler agent

Services a single entry-point:

- creates an incoming socket for specified IP-address and TCP-port;
- accepts new connections for that entry-point (stops accepting when there are too many parallel connections);
- reads initial data from accepted connection, detects the client's protocol;
- handles incoming commands from client and establishes outgoing connections to target hosts;
- transfers data between clients and target hosts.

Why an agent for entry point?

We had plenty of choices:

1. An agent that serves a group of entry points with all related connections.
2. An agent that serves single entry point and all related connections.
3. An agent that serves single entry point and an agent that serves a pair of incoming/outgoing connections.
4. An agent that serves single entry point, an agent for incoming connection, an agent for outgoing connection.
5. An agent that serves a group of entry points, an agent for incoming connection, an agent for outgoing connection.
6. ...

We decided to have an agent for entry point...

...that serves also all related connections (incoming to that entry point, and outgoing from that entry point).

The main reason: *simplicity*.

It's easy to delete entry point when it is removed from the config.

No separate agents for in/out connections

Too many agents in an application is not a good thing. It's hard to monitor and debug program with 40K live agents inside.

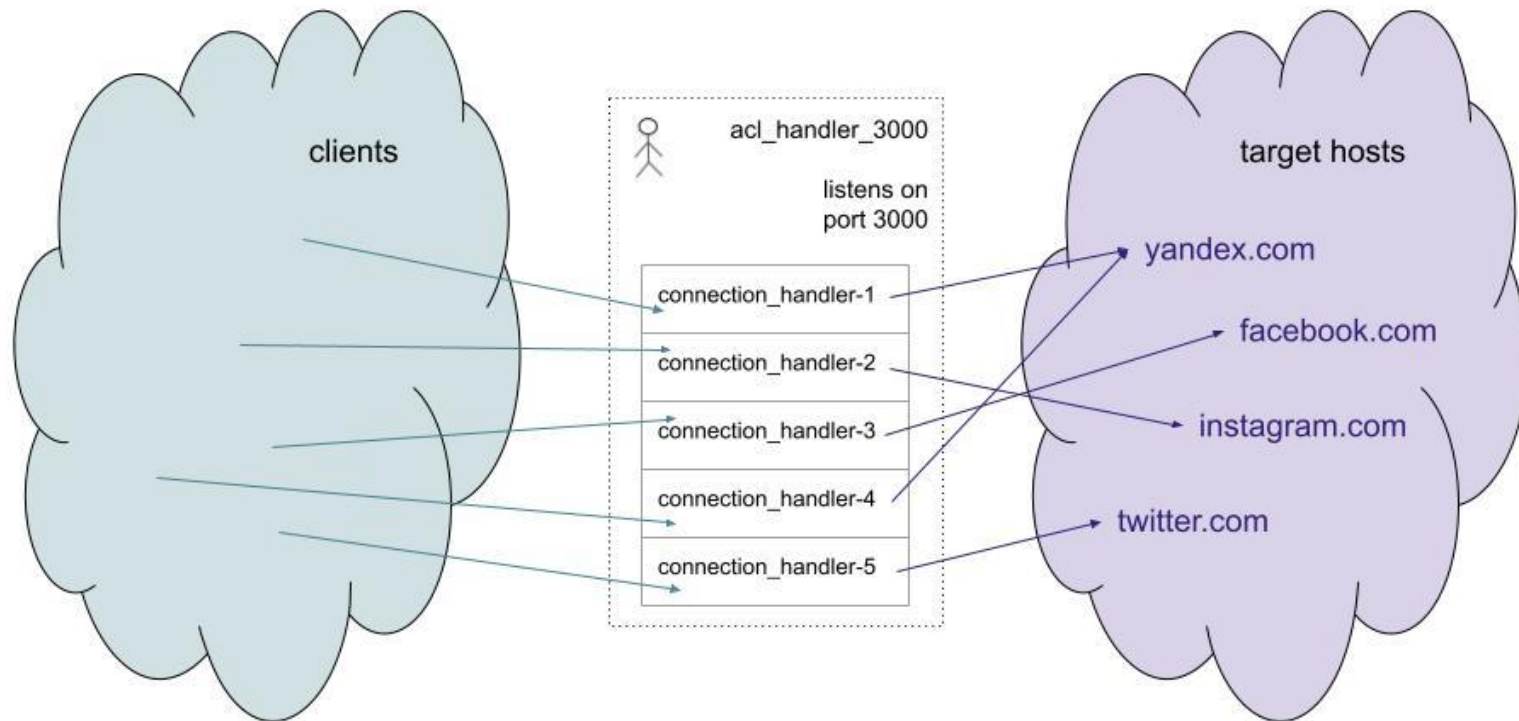
Creation and deletion of agents is more expensive operation than creation and deletion of more lightweight objects that we call `connection_handler`.

So in *arataga* an `acl_handler` agent owns many `connection_handler` objects.

An instance of `connection_handler` owns a pair of in/out connections.

`connection_handler` is created when `acl_handler` accepts new connection and is destroyed when that connection is being closed.

acl_handler's picture



Why there are several authenticator/dns_resolvers?

We have a pair of authenticator/dns_resolver agents on every io_thread.

It's an attempt to improve locality.

An acl_handler agent on an io_thread interacts only with authenticator and dns_resolver agents from the same io_thread.

There is no need to exchange any data between io_threads to serve client requests.

Distribution of updated configs/user-lists

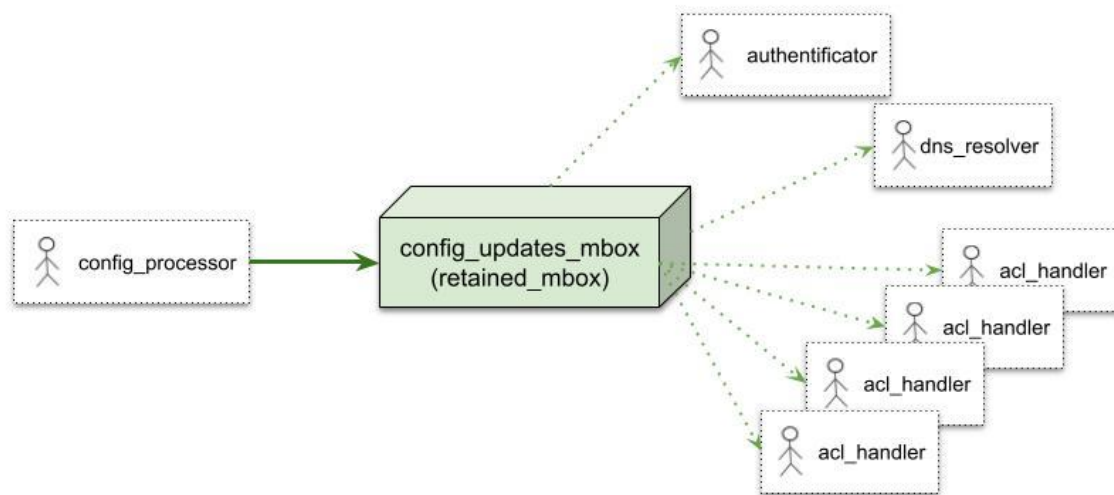
`config_manager` handles updated configs and distributes new info to all interested agents (`acl_handlers`, `authenticators`, `dns_resolvers`).

`user_list_manager` handles updated user-list and distributes new info to all authenticator agents.

That distribution is performed by `SObjectizer`'s many-producer/many-consumer message box.

Config updates distribution scheme

config_managers sends updates to config_updates_mbox and those updates are received by all subscribers of that mbox:



retained_mbox is used for config updates

A special kind of message box is used as config_updates_mbox: retained_mbox from the so5extra companion project.

That type of mbox holds a copy of the last message sent.

When a new subscriber arrives, that last copy is being automatically sent to it.

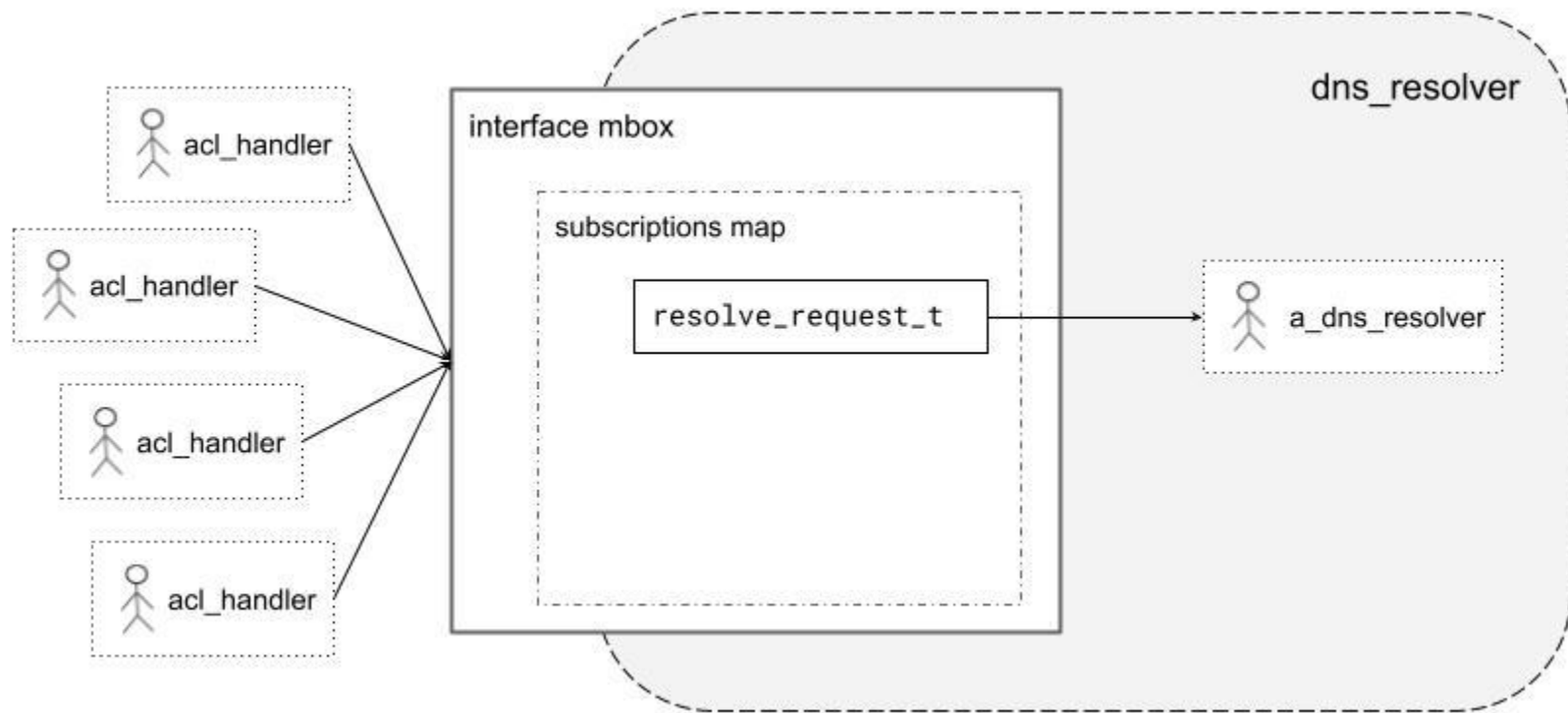
That way, new acl_handler agents automatically get the latest configuration.

dns_resolver is a coop of several agents

We rewrote interaction with DNS name servers in v.0.4 and since then there are several agents that play the role of dns_resolver.

All other agents in arataga didn't see that change because all interactions are done via separate interface mbox.

The old scheme of dns_resolver's internals



acl_handlers know only interface mbox

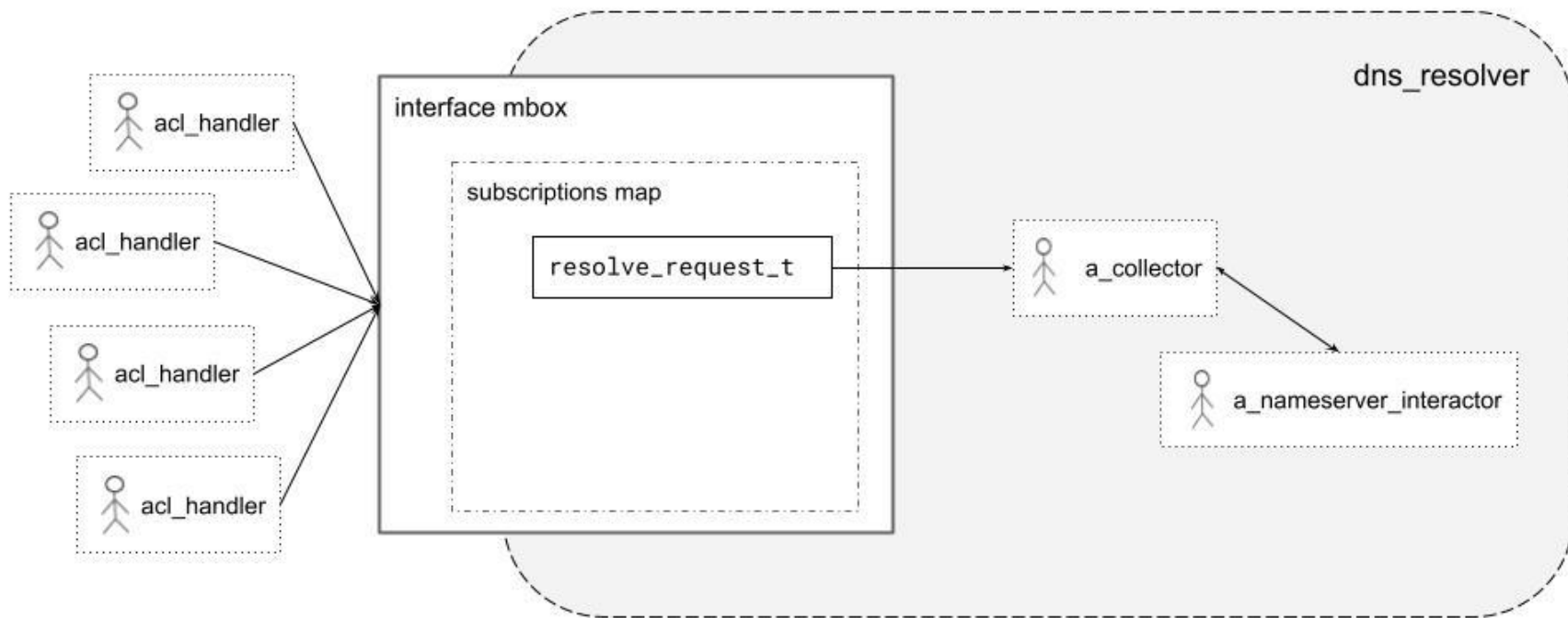
All acl_handler-agents know only the interface mbox of dns_resolver part.
No one knows what's behind that mbox.

That allowed us to replace a_dns_resolver with a couple of new agents:

- a_collector that collects incoming resolve requests, queues them, holds the results cache;
- a_nameserver_interactor that performs actual interaction with name servers using UDP protocol.

No changes were made for acl_handler during that refactoring.

Simplified new scheme of dns_resolver's internals



There are two a_collector actually...

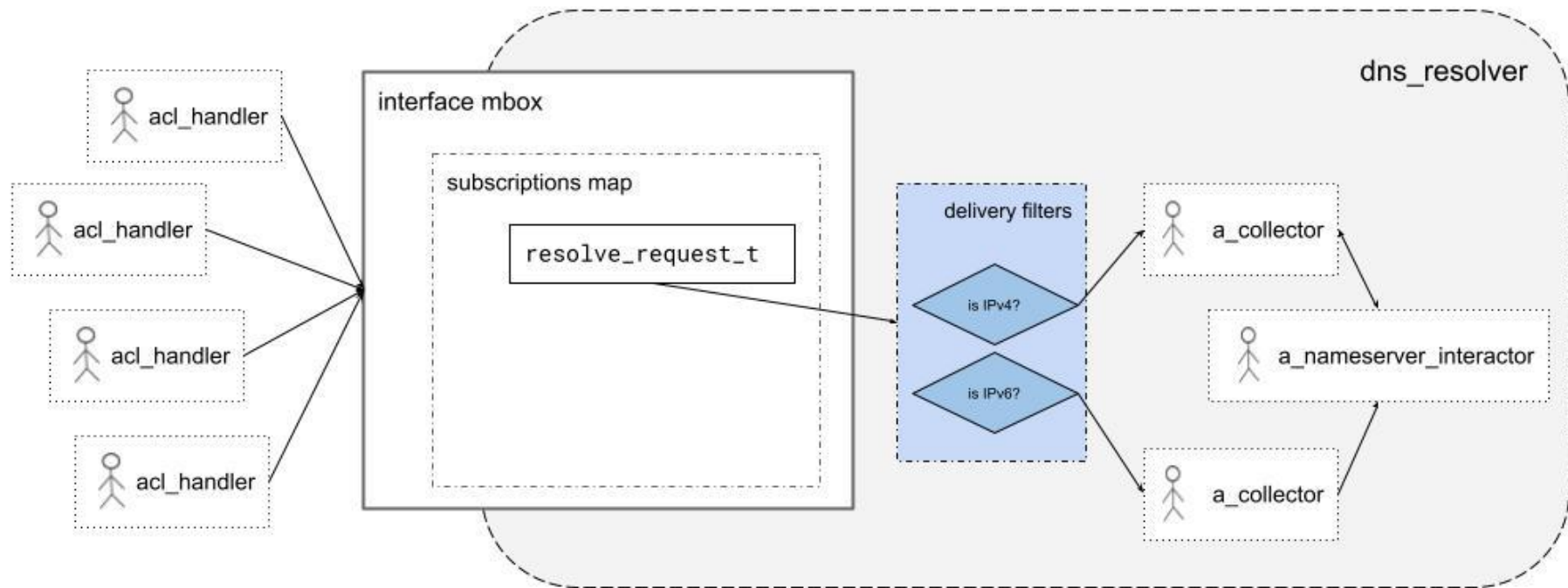
Actually, every dns_resolver part holds two a_collector agents.

One handles requests for IPv4 addresses, another for IPv6.

But they are subscribed to a single source of resolve_request_t messages.

The separation of incoming message flows is done via SObjectizer's delivery filters.

Actual new scheme of dns_resolver's internals



RESTinio's related part

How RESTinio is used in
arataga?

RESTinio in arataga: three use cases

We'll see three use cases for RESTinio in *arataga*.

One is obvious.

The second is not so.

The last is very surprised even for us.

An obvious use case: admin HTTP-entry

Admin HTTP-entry in *arataga* is implemented via RESTinio.

This entry-point accepts and handles GET and POST requests.

Every request should have a special Arataga-Admin-Token header field.

Admin HTTP-entry: launching

An instance of RESTinio server is launched by using `restinio::run_async...`

Admin HTTP-entry: launching

```
struct server_traits_t : public restinio::default_traits_t
{
    // There are only three handlers in the chain:
    // - checks for admin-token;
    // - checks for content-type for POST-requests;
    // - actual handling.
    using request_handler_t = restinio::sync_chain::fixed_size_chain_t< 3>;
};
```

Admin HTTP-entry: launching

```
[[nodiscard]] running entry handle t start entry(
    asio::ip::address entry ip, std::uint16_t entry port,
    std::string admin_token, requests_mailbox_t & mailbox )
{
    auto processor = std::make_shared< impl::request_processor_t >( mailbox );

    auto server = restinio::run async(
        restinio::own io context(),
        restinio::server settings_t< impl::server_traits_t >{}
            .address( entry ip )
            .port( entry port )
            .request handler(
                impl::make admin token checker( std::move(admin_token) ),
                impl::make content type checker(),
                [handler = std::move(processor)](auto req ) {
                    return handler->on_request( std::move(req) );
                } ),
        1 ); // Just one worker thread.

    return std::make_unique< impl::actual_running_entry_instance_t >( std::move(server) );
}
```

Admin HTTP-entry: interaction with agents

RESTinio server doesn't know about agents.

Agents don't know about RESTinio.

Interaction is performed via two special interfaces...

Admin HTTP-entry: interaction with agents

```
class replier_t {  
public:  
    virtual ~replier_t();  
  
    virtual void reply( status_t status, std::string body ) = 0;  
};
```

Admin HTTP-entry: interaction with agents

```
class requests_mailbox_t {  
public:  
    virtual ~requests_mailbox_t();  
  
    virtual void new_config( replier_shptr_t replier, std::string_view content ) = 0;  
  
    virtual void  
    get_acl_list( replier_shptr_t replier ) = 0;  
  
    virtual void new_user_list( replier_shptr_t replier, std::string_view content ) = 0;  
  
    virtual void get_current_stats( replier_shptr_t replier ) = 0;  
  
    ...  
};
```

Admin HTTP-entry: interaction with agents

REStinio's part has an actual implementation of `replier_t` interface.

SObjectizer's part has an actual implementation of `request_mailbox_t` interface.

Admin HTTP-entry: interaction with agents

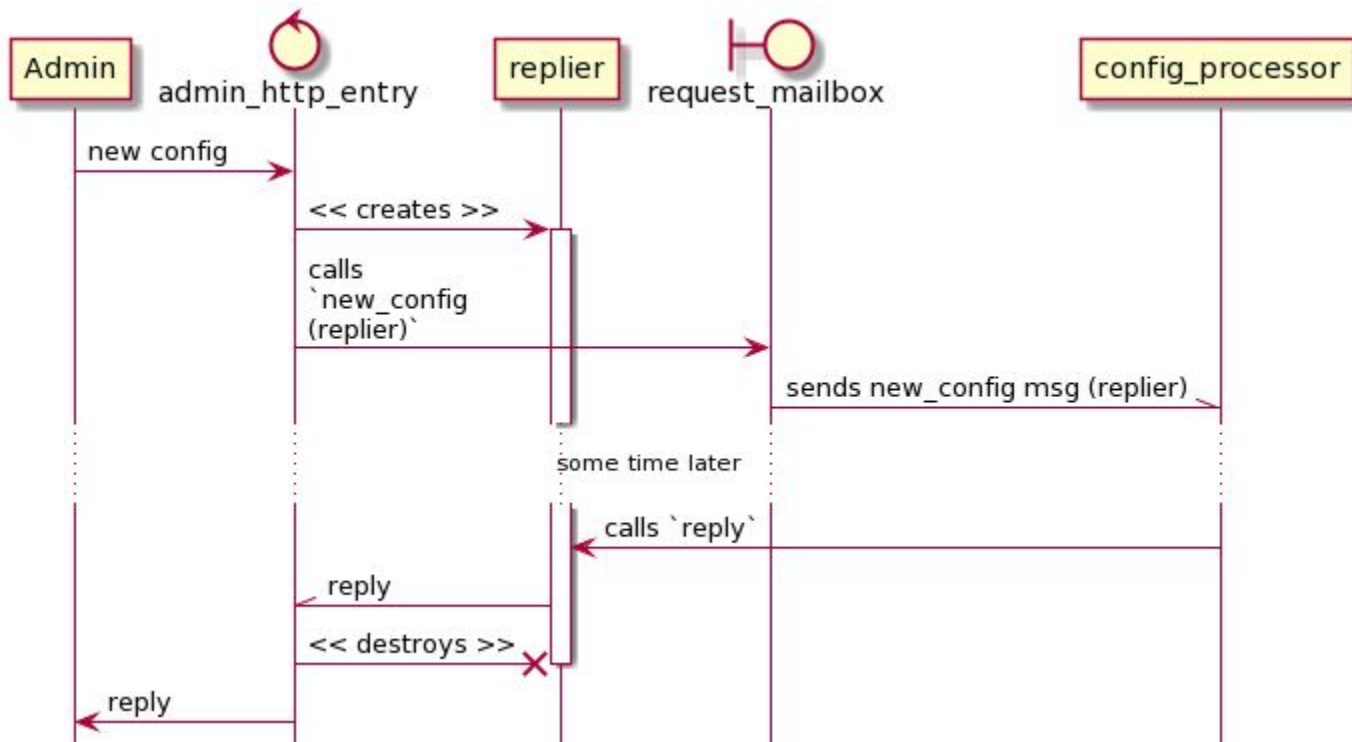
When HTTP-entry accepts a new request it creates an instance of `replier_t` and calls appropriate `request_mailbox_t`'s method.

For example, when a new config is received the `new_config()` method is called.

The actual implementation of `new_config()` send a message to the `config_processor` agent.

The `config_processor` handles new config and calls `replier_t::reply()` method.

Admin HTTP-entry: interaction with agents



Not obvious, but expected use case: HTTP-headers

We didn't plan to use RESTinio for serving incoming HTTP connections.

It's because the current version of RESTinio loads the whole request into the memory and only then allows to handle the request.

That approach can't be used in proxy where a lot of large incoming requests have to be handled without affecting the performance and resource usage.

But we hoped that some parts of RESTinio could simplify our work...

Not obvious, but expected use case: HTTP-headers

REStinio has tools for working with HTTP header fields.

Those tools were reused in *arataga*.

Not obvious, but expected use case: HTTP-headers

RESTinio's `http_header_fields_t` container is used for holding headers:

```
struct request_info_t
{
    //! HTTP-method of the request.
    /*!
     * It is stored here to be easily accessible.
    */
    http_method m_method;

    //! The value of request-target from the start-line.
    std::string m_request_target;

    //! Extracted HTTP header fields from the incoming request.
    restinio::http_header_fields_t m_headers;

    ...
}
```

Not obvious, but expected use case: HTTP-headers

Not only for holding headers, but also for handling them:

```
// If there are more than one Host header fields then the request
// should be rejected. So count the fields.
std::optional< std::string_view > opt_host;
std::size_t host_occurrences{ 0u };

m_request_info.m_headers.for_each_value_of(
    restinio::http_field_t::host,
    [&]( std::string_view value )
    {
        ++host_occurrences;
        if( 1u == host_occurrences ) { opt_host = value; }

        return restinio::http_header_fields_t::continue_enumeration();
    } );
```

Not obvious, but expected use case: HTTP-headers

RESTinio's `http_field_parsers` are used for parsing headers:

```
#include <restinio/helpers/http_field_parsers/authorization.hpp>
#include <restinio/helpers/http_field_parsers/basic_auth.hpp>
...
using namespace restinio::http_field_parsers;
const auto auth_value_result = authorization_value_t::try_parse( *opt_proxy_auth_value );
if( !auth_value_result )
    return username_password_extraction_failure_t{
        make_error_description( auth_value_result.error(), *opt_proxy_auth_value )
    };
...
auto basic_auth_result = basic_auth::try_extract_params( auth_value );
if( !basic_auth_result )
    return username_password_extraction_failure_t{
        fmt::format( "basic-auth param extraction failed: {}" ,
            static_cast<int>(basic_auth_result.error()) )
    };
};
```

Totally unexpected use case: config parsing

We have to parse a couple of config files with custom syntax.

That syntax wasn't too complex in general, but the standard C++ library totally lacks any useful tools for such a task :(

Moreover, we wanted to make the syntax more user-friendly. For example, we wanted to have an ability to write like that:

```
acl.io.chunk_size 16kib # Instead of 16384
```

```
timeout.authentication 12s # Instead of 12000
```

```
dns_cache_cleanup_period 6min # Instead of 360
```

Totally unexpected use case: config parsing

We solved that task by using RESTinio's `easy_parser` that is used in RESTinio for parsing HTTP header fields.

As an example we'll show how values like 1min, or 25s, or 125ms are parsed...

An example of RESTinio-based parser (1)

Factory for parsers of time-out values:

```
/*!
 * @brief A producer for easy_parser that extracts time-out values
 * with possible suffixes (ms, s, min).
 */
[[nodiscard]] static auto timeout_value_p()
{
    struct tmp_value_t
    {
        std::int_least64_t m_count{ 0 };
        int m_multiplier{ 1000 };
    };

    using namespace restinio::http_field_parsers;
    using std::chrono::milliseconds;
```

An example of RESTinio-based parser (2)

```

return produce< milliseconds >(
    produce< tmp_value_t >(
        non_negative_decimal_number_p< std::int_least64_t >() >> &tmp_value_t::m_count,
        maybe(
            produce< int >(
                alternatives(
                    exact_p( "min" ) >> just_result( 60'000 ),
                    exact_p( "s" ) >> just_result( 1'000 ),
                    exact_p( "ms" ) >> just_result( 1 )
                )
            ) >> &tmp_value_t::m_multiplier
        )
    )
    >> convert( [] ( const auto tmp ) { return milliseconds{tmp.m_count} * tmp.m_multiplier; } )
    >> as_result()
);
}

```

An example of RESTinio-based parser (3)

A usage of that parser:

```
///Handler for `dns_cache_cleanup_period` command.
class dns_cache_cleanup_period_handler_t : public command_handler_t
{
public:
    command_handling_result_t try_handle(
        std::string_view content, config_t & current_cfg ) const override
    {
        return perform_parsing( content, parsers::timeout_value_p(),
            [&]( std::chrono::milliseconds v ) -> command_handling_result_t {
                if( std::chrono::milliseconds::zero() == v )
                    return failure_t{ "dns_cache_cleanup_period can't be 0" };

                current_cfg.m_dns_cache_cleanup_period = v;
                return success_t{};
            } );
    }
};
```

Epilog

Unfortunately, arataga wasn't stress-tested enough

Because the customer lost interest in the project, the *arataga* wasn't stress-tested under the huge loads.

Only a few stress tests were performed.

In config and load dependency, *arataga* used from 1.5 to 4 times less CPU consumption than the customer's old proxy server under the same workload.

The current status

The active development of *arataga* was suspended in 2020.

We ourselves consider *arataga* to be an excellent testing ground for the approbation of new versions of SObjectizer and RESTinio in real-life conditions.

So we make some changes to *arataga* from time to time when we have some new idea.

We also fix issues when someone reports them. If you find a problem in *arataga* feel free to open an issue on GitHub.

Our own impressions: RESTinio

The use of RESTinio in *arataga* gave us some ideas for new RESTinio's features.

One of them, chain of synchronous handlers, was implemented and available since RESTinio-0.6.13.

Others still wait their time...

Our own impressions: SObjectizer

Usual routine, nothing exciting, no challenges.

We took SObjectizer, wrote several agents, and that's all.

SObjectizer just works. There weren't any SObjectizer's or multithreading-related issues.

References

<https://github.com/Stiffstream/arataga>

<https://github.com/Stiffstream/sobjectizer>

<https://github.com/Stiffstream/so5extra>

<https://github.com/Stiffstream/restinio>

That's all

Thanks!