telegram state machine design doc

Document #: telegram state machine v1
Date: 2022-02-07
Project: Programming Language C++
Audience: dev team
Reply-to: Marco Rubini
<malorubi@outlook.com>

Contents

1 Introduction

This document defines a library component with the following capabilities:

1. Abstraction of resource ownership of a single chat, group or channel.
2. Define commands with optional parameters passed as space delimited strings
3. Define restricted commands that follow a hierarchical permission system
4. Allow complex processing of a command
5. Given a command handler, we should be able to:
6. Send messages containing formatted text and keyboards
7. Ask the user for further interactions, like sending a message or pressing a button
8. Handle further interactions in the context of the same command
9. Sending sensitive information
10. Auto-delete messages with sensible information after a fixed timeout
11. Allow the user to easily delete a single received message
12. Allow timing out of user interactions
13. Allow sending messages (notifications) to registered users/groups/channels

2 Discussion

This component depends on the state machine component *(see document state machine v1)*

The telegram state machine (TSM) extends the concepts provided by the state machine (SM) component, and provides some implementation defined states and transitions that handle common interactions.

A TSM only exposes the functionalities necessary to interact with a single telegram chat, group or channel. Another component will handle collecting and dispatching events to the exact TSM capable of handling them.

3 Context

A TSM Context extends a SM Context, and also provides methods to interact with the chat/group/channel.

```cpp
namespace forest::tsm
{
  struct send_message_result
  {
    std::int64_t id;
  };

  struct button
  {
    std::string id;
    std::string description;
  };

  template<class T>
  concept Context = forest::sm::Context<T>
    && requires(T context, std::string text, std::vector<button> buttons, std::int64_t id)
  {
    // send_message overload 1
    { context.send_message(text) } -> std::same_as<send_message_result>;

    // send_message overload 2
    { context.send_message(text, buttons) } -> std::same_as<send_message_result>;
    // delete_message overload 1
    { context.delete_message(id) } -> std::same_as<void>;

    // edit_message overload 1
    { context.edit_message(id, text) } -> std::same_as<send_message_result>;

    // edit_message overload 2
    { context.edit_message(id, text, buttons) } -> std::same_as<send_message_result>;
  };
};
```

3.1 send_message

The method `send_message` has multiple overloads.

Each overload returns a `send_message_result`, a type that contains information about the message just sent.

The request is blocking and can throw if a network error occurs, including a network timeout.

Overload descriptions:

1. sends a text message.
2. sends a text message with a basic keyboard.

3.2 delete_message

The method `delete_message` deletes a previously sent message.

The request is blocking and can throw if a network error occurs, including a network timeout.

Overload descriptions:

1. deletes a message with the given id.

3.3 edit_message

The method `edit_message` modifies a previously sent message.

The request is blocking and can throw if a network error occurs, including a network timeout.

Overload descriptions:

1. edits a message with the given id by replacing its text component.
2. edits a message with the given id by replacing its text and keyboard components.

4 State and Transition

TSM states and transitions concepts are the same as SM states and transitions.

5 Command

TSM commands are transitions with additional `prefix` and `description` methods.

```
namespace forest::tsm
{
  template<class T, class Context, class State, class Event>
  concept Command = forest::sm::Transition<T, Context, State, Event>
    && requires(T command, Context context, State state, Event event)
  {
    { command.prefix() } -> std::convertible_to<std::string>;
    { command.description() } -> std::convertible_to<std::string>;
  };
}
```

6 AuthorizedCommand

AuthorizedCommand extends the Command concept by exposing a `required_privileges` method, which returns a collection of privilege names.

```
namespace forest::tsm
{
  template<class T, class Context, class State, class Event>
  concept AuthorizedCommand = forest::tsm::Command<T, Context, State, Event>
    && requires(T command, Context context, State state, Event event)
  {
    { command.required_privileges() } -> std::same_as<std::vector<std::string>>;
  };
}
```

7 Authorization subsystem

The authorization system simplifies defining and checking privileges before triggering command transitions.

A command can require one or more privileges, identified by their name, a string of lowercase alphabetical characters and underscores.

7.1 Grant

A user can be granted any privilege using the `/grant user privilege_name` command, implemented by a predefined transition handled by TSM.

The `/grant` command is itself an AuthorizedCommand, which requires the `can_grant` privilege.

7.2 Revoke

A user can be revoked any privilege using the `/revoke user privilege_name` command, implemented by a predefined transition handled by TSM.

The `/revoke` command is itself an AuthorizedCommand, which requires the `can_revoke` privilege.

7.3 Groups

Privilege groups, not to be confused with Telegram Groups, allow hierarchical grouping of privileges.

7.3.1 Group Add

A user can create a new group using the `/groupadd group_name` command, implemented by a predefined transition handled by TSM.

The `/groupadd` command is itself an AuthorizedCommand, which requires the `can_groupadd` privilege.

7.3.2 Group Del

A user can delete an exhisting group using the `/groupdel group_name` command, implemented by a predefined transition handled by TSM.

The `/groupdel` command is itself an AuthorizedCommand, which requires the `can_groupdel` privileges.

7.3.3 Group Grant

A user can grant a privilege to a group using the `/groupgrant group_name privilege_name` command, implemented by a predefined transition handled by TSM.

The `/groupgrant` command is itself an AuthorizedCommand, which requires the `can_groupgrant` privilege.

7.3.4 Group Revoke

A user can revoke a privilege to a group using the `/grouprevoke group_name privilege_name` command, implemented by a predefined transition handled by TSM.

The `/grouprevoke` command is itself an AuthorizedCommand, which requires the `can_grouprevoke` privilege.

7.3.5 Group List

A user can retrieve a list of all created groups using the `/grouplist` command.

The `/grouplist` command is itself an AuthorizedCommand, which requires the `can_grouplist` privilege.

7.4 Modify a User's groups

Every user has a (possibily empty) list of groups associated.

7.4.1 UserGroupAdd

A user can add another user (possibly theirselves) to a group using the `/usergroupadd username group` command.

The `/usergroupadd` command is itself an AuthorizedCommand, which requires the `can_usergroupadd` privilege.

7.4.2 UserGroupDel

A user can add another user (possibly theirselves) to a group using the `/usergroupdel username group` command.

The `/usergroupdel` command is itself an AuthorizedCommand, which requires the `can_usergroupdel` privilege.

7.4.3 UserGroupList

A user can list another user's groups using the `/usergrouplist username` command.

The `/usergrouplist` command is itself an AuthorizedCommand, which requires the `can_usergrouplist` privilege.

7.5 Special groups

Some groups are provided by default, they can't be removed.

7.5.1 Admin group

The Admin group provides all privileges.

The Admin group's privileges can't be revoked.

The chat/group/channel owner is automatically added to the Admin group.

Only members of the Admin group can add/remove users to/from the Admin group.

7.6 User database

Information required for the authorization system is stored in a SQLite database, in a table specific to a chat/group/channel.

Table schema:

CREATE TABLE Authorization(chat_id string PRIMARY KEY, user_id string PRIMARY KEY, group_list string NOT NULL, privilege_list string NOT NULL);

Groups and privileges are stored in comma-delimited strings.

7.7 Privilege checking control flow

A library user doesn't have to handle privilege checks at all.

When a command that modifies the Authorization table is issued, TSM checks the invoking user's privileges and admin status and either performs the command or responds with a predefined error message.

When an AuthorizedCommand is issued, TSM checks that the privileges required by the command (retrieved using the `required_privileges` method), are owner by the invoking user, and either performs the command's action or responds with a predefined error message.

(todo: allow a library user to change the message or action performed on privilege error)

8 Sensitive messages

When sending sensitive information, a library user can program the deletion of a message in the future.

8.1 Proposal 1

Allow scheduling the firing of events in the future, by adding an overload to the `fire_event` method of a context.

To delete a message in a future, a library user can store the id of the message just sent, and schedule its deletion.

Message deletion events are handled by an implementation-defined transition that does not change the internal state of TSM.

```
// Example
struct some_state
{
  template<forest::tsm::Context T>
  void on_entry(T context)
  {
    using std::chrono::milliseconds,
    using forest::tsm::event_request_message_deletion;

    std::int64_t id = context.send_message("Hello").id;
    context.fire_event(event_request_message_deletion{id}, milliseconds(5000));
  }
};
```

9 Notification subsystem

TODO