1. With telegram, users can share their current locations in chats. We were interested in this function. Thus, we worked on how sharing location was implemented in the project:

First, we noticed there was a class called *LocationSharingService*. We thought it was related to the location sharing. Thus, we checked it. However, it only worked as a service when multiple users are sharing their current location at the same time. We still need to find the function for sending a location as message.

Thus, we checked *LocationController*. It was used to control the current location when sharing. It is also not related to sending location.

Considering that the location might be sent as a message, we went to *MessageObject* class. In that class, we found a method called *isLocationMessage().* It is called when sending a location. Each time sending a location, it is used to check whether the message is a location.

Then we tried to find the usage of the *isLocationMessage().* We found it in the *MessageObject* class. It was called by isLocation() in *MessageObject* class. Then the *isLocation()* method was called by *updateButtonState()* method in *ChatMessageCell* class, which was related to app UI. This way comes to an end.

Thus, we tried to find the declaration of *MessageObject*. First, we found a class called *LocationActivity*. This class was used to create the map view for the app. But not for sending location message.

Then we found a class called *SendMessagesHelper*. In this class, a method called *sendCurrentLocation()* initiated a location *MessageObject* and send it.

In *MessageObject*, the location data is retrieved form *LocationProvider*, thus we go to *LocationProvider*. It is an inner class in *SendMessagesHelper* to retrieve location from GPS listeners.

Thus, the whole process should be like: The *LocationProvider* retrieve location data from GPS listeners, then make it a *MessageObject*. The message is sent and *LocationActivity* is used to create the map view. If users are sharing locations in real-time, *LocationSharingService* and *LocationController* are used to coordinate real-time location data form multiple users.

Folder	File	Method	Relevant	Relevant how?	Confidence(1-5)	Notes
/messanger	LocationSharingService.java	onCreate	Y	Multiple users share location in the same map.	5	Not used when sending location
/messanger	LocationController	getInstance()	Y	Return a LocationController instance	5	
/messanger	LocationController	update	N	/	1	Update location
/messanger	MessageObject	isLocationMessage	Y	Called when sending location or sharing location	3	Check if the message is a location
/messanger	MessageObject	isLocation	N	/	1	Called for ui updating
/ui	LocationActivity	createView()	Y	Creat map view when using location	5	
/messanger	SendMessagesHelper	sendCurrentLocation()	Y	Create Location message	5	
/messanger	LocationProvider	/	Y	Retrieve location data	5	

Folder	File	Method	Why	Priority (1-5)	Notes
/messanger	LocationSharingService.java	onCreate()	Named LocationSharing; onCreate is called when activity start	5	
/messanger	LocationController	getInstance()	Called in LocationService.java to get location	5	
/messanger	LocationController	update()	Update current location when sharing.	1	
/messanger	MessageObject	isLocationMessage()	Called when sending a location message	3	
/messanger	MessageObject	isLocation()		3	
/ui	LocationActivity	createView()	Using MessageObject.isLocation()	5	
/messanger	SendMessagesHelper	sendCurrentLocation()	Declaring MessageObject and using Location as input	5	
/messanger	LocationProvider	/	Used to retrieve location data	5	

2. To find classes relevant to feature Block User, we started off by simply searching for blockuser, and luckily a function with exactly the same name *blockUser* was among the results.

After inspecting the function, we concluded that it indeed did what its name suggested.
Specifically, if first gets the id of current user, and puts the id into list of blocked ids. It also invokes different subsequent methods depending on whether the user to be blocked is a bot.

Among methods invoked by *blockUser, getUser* simply returns the current user id. Thus although it is a relevant component, it does not lead us to subsequent functions.

Thus we went back to check in *MediaDataController.java* for the *removeInline* method which was called when what was supposed to be blocked was a bot. It removes the bot that user want to block from current list of bots. Having observed how *removeInline* works, we can (almost) safely assume that *removePeer()* work in the same way to remove a non-bot user from a maintained list.

TL_contacts_block, which appeared in search for keyword block, seemed to be a relevant class. When the inner id field of the object is set (in this case to the current user), it can invoke subsequent method to serialize id to byte stream. Further invocation of these methods was not found anywhere, so we chose to return to the search for keyword.

ShowBlockAlert was certainly a relevant component since it was a ui controller. It defines buttons and pop-up warnings (e.g.: “Are you sure you want to block xxx?”) related to user-blocking actions.

canBlockUsers and *canUserDoAction* were checks for user authorizations for blocking and current status of current user.

We also checked out the class *PrivacyUserActivity* because it extensively used a variable named “*blockedUsersActivity*”, which marks relevance. The activity file controls ui of parts of privacy settings, including management of blocked users.

Since blocking was a state that needed to be specifically marked, we assumed that most of the relevant functions should contain the keyword “block”. Thus our discovery mainly oriented from the search for the keyword.

Folder	File	Method	Relevant?	How	Confidence(1-5)	Notes
messenger	MessagesController.java	blockUser(int user_id)	Yes	Adds specified user to blocked list, updates count	5	
messenger	MessagesController.java	getUser	Yes	Gets the current user from a concurrentHashMap	5	Dead end. Go back.
messenger	MediaDataController.java	removeInline	Yes	When blocked, removes the bot from the arraylist of bots.	5	Dead end. Go back.
messenger	TLRPC.java		Yes	Looks like the user to be locked is registered to the class.	3	Go back
ui	DialogOrContactPickerActivity.java	showBlockAlert	Yes	Ui related logics.	5	
messenger	ChatObject.java	canBlokcUser	pending(yes)		3	Leads to next method
messenger	ChatObject.java	canUserAction	Yes	Blocking authorization depends on *canUserDoAdminAction*	4	Should not need to look further
ui	PrivacyUserActivity.java		Yes	Updates and populates user status, including whether it is blocked	3	

Folder	File	Method	Why	Priority (1-5)	Notes
messenger	MessagesController.java	blockUser(int user_id)	Appears to be the most relevant one by name.	5	Quite obvious
messenger	MessagesController.java	getUser	Called in blockUser	5	Probably won't lead us anywhere, but necessary for the functionality.
messenger	MetaDataController.java	removeInline	Called when user to be blocked is a bot	4	We would want to know what MediaDataController is.
tgnet	TLRPC.java	*Class*	A new instance initialized in blockUser method	5	
ui	DialogOrContactPickerActivity.java	showBlockAlert	Related to the block interface	4	
messenger	ChatObject.java	canBlockUsers	Looks like a check on authority of user action	4	
messenger	ChatObject.java	canUserDoAction	Need to check this to determine relevance of this and previous method	3	
ui	PrivacyUserActivity.java	*Class*	Extensive use of field "blockedUsersActivity"	3	