Dialogue Stacks

Author: Ian Wang

Overview

A more streamlined method of developing bot applications, called **dialogue stacks**, are implemented in the API as well. A *Dialogue* defines how the bot server handles the incoming events of a certain user, and each user is allocated their own dialogue stack. Dialogues can maniplate the stack from within itself, allowing extensive customization on the behaviour within each dialogue.

Benefits

The biggest benefit of dialogue stacks is that it is easy to reason dialogue flow. Each dialogue can be thought of as a 'page' in the sense of a traditional app. Each dialogue handles the inputs differently, just like how each page has different functions for the user to interact with. The stack implementation allows user to easily return to a 'previous' dialogue, just like how using the back button in an app or website will lead you to where you were before. Once a traditional app flow diagram is drawn out, it is essentially trivial to convert it to a dialogue flow using dialogue stacks.

Another big benefit is the implicit separation of user states. Since each user gets their own dialogue stack, then we know that variables stored inside a dialogue is individual to each user. Thus, storing temporary data from users becomes as trivial as creating a new variable inside a dialogue to keep track of whatever it is that needs to be kept track of.

Issues

One concern that could be raised with the implementation of dialogue stacks is performace. Since each user is allocated a dialogue stack, the program has to allocate extra memory for each user as well as pass through some extra levels of indirection. However, in most cases, these concerns will not have too significant of an impact as long as the dialogue flow is well designed.

First, to address the issue of dialogue stacks requiring too much memory, a deeper look into the implementation of dialogue stacks is required. Each Dialogue contains 4 object references, which is 4 * 32 bits in most JVMs, which means that the dialogue object itself takes up 16 bytes of data. The dialogue stack itself also contains an object reference, which is another 32 bits (4 bytes) of data, so each user takes up at least 20 bytes of memory to store its dialogue stack. Since everything is stored in a Hashmap, another object reference is stored alongside the userId String. Since the userId is stored anyways, its cost will not be counted towards the cost of Dialogue Stacks. Thus, each user takes an extra 24 bytes of memory total at minimum.

If the bot server were to handle 1000 users, it would need roughly 24 kilobytes of memory to store the dialogue stacks of each user, which is easily handled by any modern machine. Even at 1,000,000 users, there would be 24 megabytes of memory needed to store all the dialogue stacks, which still isn't much for modern processors to handle. The problem comes when dialogue flow is badly designed and dialogue stacks start to contain too many dialogues. The average number of dialogues per stack is directly proportional to the memory used, so if there are on average 3 dialogues per stack, then we will require roughly 3 times the memory of one dialogue in each dialogue stack. Therefore, a well designed dialogue flow averaging one dialogue per user will not have a significant impact on the memory required to run the program.

Addressing the efficiency issues with the levels of indirection using dialogue stacks, only the implementation of the storing of dialogue stacks needs to be examined. Looking at the implementation without dialogue stacks, an incoming event goes through a central routing function to convert the JSON data into a Message object, and then calls the handleEvent function. With dialogue stacks, the program finds the dialogue stack of the corresponding user after converting the data into a Message object, and then calls the handleEvent function of the top dialogue of the stack.

One extra function call is small enough to not be noticable, so the biggest performance difference should be finding the dialogue stack of the user. Since a HashMap is used to store the users and their corresponding dialogue stacks, finding the correct stack in a map is constant time, the Map is big enough for there to be minimal collisions. Even with collisions, the HashMap will detect if the entry is big enough to justify using a Binary Search Tree, which will result in logarithmic time at worst.

As shown above, the performance issues of dialogue stacks don't have a significant impact, and is easily worth it for easier dialogue design and code readability.