ChatPatch

Jacc734

February 4, 2018

Abstract

This will be a technical report on the Minecraft Spigot server plugin ChatPatch. This is written so that those whom must review ChatPatch are able to do so as thoroughly and efficently as possible. It will go over the classes, methods, and variables. However, at times, it will also include deep insight into the design choices, and the base purpose of the ChatPatch plugin.

Contents

1 The ChatPatch Class

```
1 public class ChatPatch extends JavaPlugin implements Listener
```

This class is the core of the plugin as noted by the fact it extends JavaPlugin. It also implements Listener; this is for the ProtocolLib PacketAdapter that is used to change the messages contained filtered words. While I certainly believe the code could be implemented in pure bucket code, I don't think the solution would be as nice, though it may be something to consider if the impact on server resources with this attempt are unfavorable.

1.1 Declarations

```
1 private JSONParser parser;
2 private ProtocolManager protocolManager;
3 private PluginManager pluginManager;

5 private final CensorListener censorListener = new CensorListener(this);

7 public final HashMap<Player, Boolean> censorMailList = new HashMap<Player, Boolean>();
8 public final List<String> censorID = getConfig().getStringList("censored");
```

1.1.1 JSONParser++

The JSONParser is not currently used. The declaration is kept though, because it will be used to further parse the messages. Chat packets in Minecraft are layered in objects, but at the core is a JSON format.

The parser will allow us to extract the exact section that contains the message and filter it. Currently, the entire JSON string is being filtered. This could have some unexpected results, particularly if RegEx were to be used.

ChatPatch is only a simple Patch for people who want a little fix. The next plugin will depend on some of the technique in ChatPatch to build a first tier censorship plugin. The only plugin that allows players to choose their own level of censorship.

1.1.2 ProtocolManager++

The ProtocolManager is how you manage the packet manipulation. Having this allows us to add a PacketListener (or PacketAdapter), and then override a the correct functions to censor the CHAT packets coming from the server.

1.1.3 PluginManager++

The PluginManager allows us to register the listener. After registering the listener it isn't used directly again. It could easily be removed if the plugin was in its final stage. However, with the development still underway this is silly.

1.1.4 CensorListener++

The CensorListener was orginally going to contain all the listener class code, however, implementing the PacketListener outside the main function became more difficult than originally thought. So now all CensorListener does is add players logging in and out of the server to a list of users who are censor-users.

This will allow the server to know which recipients should have their incoming chats screened.

In the future, an attempt might be made to move the censoring process back into the CensorListener class entirely, however, there is no rush for that change.

1.1.5 censorMailList++

The HashMap ⟨ Player, Boolean ⟩ labeled censorMailList keeps track of all the online players that have subscribed to the censorship chat feed. It is added to, and removed from as people login and logout. This is what is checked when the plugin checks if a player should received censored messages.

1.1.6 censorID+

The List ⟨ String ⟩, censorID (should) contain UUID's that were stored in config. These UUID's are the list of all the players who enabled censoring on their accounts. Only if the player has the ChatPatch enabled when they exit the game, does the plugin save their UUID to the censorID list permanently. The censorID is populated when defined; it pulls in all the entries under the config.yml file section "censored" which is a list of censor users UUID's as a list.

1.2 onEnable()

The onEnable function performs all the standard functions that are expected from it in minecraft server plugins. First, it saves the default config; this will not overwrite an existing config, but will place a file if the config.yml is not found in the plugin's data folder.

Next, all the initializations take place for many of the declarations mentioned in the previous section including managers for the plugin and protocolLib. A parser is also added for the JSONparser.

The listener is registered, but then the real interesting stuff would begin because censorPackets() function is called, for more information on that. Read the appropriate section.

The plugin then checks the list of online players against the UUID's listed as censor users. If they are censor users they are added to the censorMail-List HashMap. In the future, I may switch to using a second list instead of a HashMap, as I am only checking for the key, or I may come up with a use for the value field. However, Hashes are faster than lists so I will have to look into that more before making a definitive choice.

After this, the executor for the plugin is set to ChatPatchCommand. Last, the server logs INFO printing the plugin was enabled.

```
1    getCommand("chatpatch").setExecutor(new ChatPatchCommand(this));
```

1.3 onDisable()

The onDisable() method saves the config with the updated list of censor users. Then logs the plugin is disabled. It is assumed that the plugin's Listener and PacketAdapter are automatically unregistered and disabled. While this is certainly standard for listeners, I had a passing concern that a PacketAdapter listener may not do this. Though, from ProtocolLib examples and other plugins I have not seen usage of commands to disable such packet listeners. So I will not concern myself with it, at least for now.

1.4 censorPackets()

This is the core of the plugin's purpose, it handles the packet interception and filtering. While I may wish to move the packet interception into its own class file at some point, I am happy with the current results, and will polish for an initial release before making any drastic changes to the code base.

1.4.1 ProtocolLib: Packet Listeners

As with most new API's there was some level of frustration when things didn't work how I expected. So I will include things that are basic, but were useful in my learning processing of interacting with ProtocolLib.

```
1    protocolManager.addPacketListener(new PacketAdapter(this, ListenerPriority.NORMAL, PacketType.Play.Server.CHAT) {...});
```

The protocolManager we used earlier must be used to add a packet listener, this is one reason I decided to keep this in the main plugin class; I detest passing lots of variables all over the place. It feels like a waste of memory space, but I know its not a big deal, so eventually I'll probably split this out.

When adding a packet listener you can create a new PacketListener, or a new PacketAdapter. I barely looked into the difference, but apparently the standard is to use PacketAdapters.

When creating a PacketAdapter you can notice, I pass the plugin (this) to attach it. I assign a ListenerPriority, I chose NORMAL, because I want to make sure there is minimal strain on the server. Though, I recently read a post where someone had a variable priority - I like the sound of that, and may work a option into the config to change priorities.

PacketType is by far the most interesting and complex parameter. I had been stumped because there is or was another method of defining what packet type you are listening before. The other method, either isn't used anymore, or requires a different application/method, as 'new PacketAdapter(...)' was much happier with the syntax/definition I provided above.

The first term is 'Play' this indicates that the server is interacting with a logged in client, a client is usually a client.

The second term is 'Server', this is an extremely important term. It tells you a lot about the packet. It is coming out of the server. This means the server has already processed the incoming packet from the client. If you want to catch

packets entering the server, before they are processed on, then you want to use 'Client'.

The third term is 'CHAT'. This a little misleading. When you think of chat you may think of what people want to communicate, but in a server's mindset this includes commands typed by players, broadcasts, anything textual that passes between the server and the client, even if the result isn't always visible (like some commands with no apparent result).

1.4.2 ChatPatch's Censor Process

This is a powerful combination of terms, and to understand them better we need to continue changing how we think about chat to how the server thinks about chats. When we send a message we expect that once it is launched we are sending to everyone. This is a lie. We send it the server. Once it reaches the server we might think it sends it to everyone, that is a half-truth.

The server sends the chat you send to everyone, to each client machine as a separate message - if you are technically inclined this may seem a silly and obvious detail, but I place emphasis on it because it explains the power of the ChatPatch plugin.

Since the server must relay the chat message to each client machine individually, it is not necessary to filter the incoming messages like many other censor plugins and people think.

If you check which client the server is about to send a message too, then you can see if they request censoring, if so you can modify the message just for them. They are the only one to see the change made.

This allows for infinitely (jk, per-player) customization of censoring. However, while having one filter-words list per a player would be easy to implement, it is a little over kill. Instead more generalized solution is to pretend that all users who opt for censoring are a "group". By tracking this "group" of players who request censoring, we can use their UUID's to apply the same filter to their messages.

This is what ChatPatch does. Note this still has great weakness; which is okay. As the first ground rule of any censor-solution, is that no censorship solution is ever perfectȦChatPatch does some manipulation following the Vilfredo Paret principle by Joseph Juran *solve 80% of the problem with 20% of the effort* (or in my case 35% effort).

But at the end of the day, there is still a gap in coverage. ChatPatch only does exact word matching. Using tricks like putting everything in lowercase before checking the message for filter-words (currently implemented), and replacing symbols/numbers with best-match letters (future work) bring the problem well down by base digits. $52^{\text{input-text}}$ to $26^{\text{input-text}}$ for alphabet inputs in combinational calculations, but it still isn't enough.

The next step is Reg-Ex, however, something more robust is just that more robust. Heavier, a tank like structure, compared to the quick-strike capabilities of my current plugin.

This is why I plan to make my RegEx censor plugin separate from this one. Right now my plugin parses entire outgoing messages and commands that individual players are receiving. If I were to implement more computationally expensive operations I would want to optimize the message censorship to actually be group functionality. This in turn would require a massively complex and in-depth json parsing of all messages to identify packets that all players in the censor group would receive, and thus just clone one censored packet and send it to the censor group. However, personal messages and such would still need personalized filtering, not to mention more complex parsing. The parsing would change from server to server, as other plugins would increasingly become a factor in how the json packets should be parsed.

This is why I use a blanket filtering across the json packets that match outgoing from server to player under the CHAT protocol. This allows for protection of /msg, /tell, /me, and any other communication command. Since the server is censoring its own communication protocol for messages aheaded to the censor subscribers, nothing makes it past.

Not even the server can send a filter-word to the client without it visually be translated to the specified blot character ('.').

1.4.3 @Override onPacketSending(PacketEvent) Methods

Now, lets jump back more concrete, back to the code. When you .addPacketListener(...) you have to define the PacketAdapter as aforementioned, and define it because it is an abstract class otherwise.

When creating the PacketAdapter class, you will likely override one of two (or possibly both). At least one must be overriding, which one depends on application. If in your PacketType parameter you chose Server, then you know you are dealing with packets that are about to be sent by the server to the player (any functionally related to those packets on the server's side has likely completed, unless its a multistage process, chat is not), this means you need to @Override the onPacketSending(...) function because you are going perform actions on the packets the server is about to send.

However, if you chose Client. Then this means you are intercepting the packets as soon as they reach the server from the client. This means before the Server has a chance to parse and use the packet from the client you have a chance to manipulate it. In this case you must @Override onPacketReceiving(...), as the server is receiving the packet from the client and performing some action before processing the event packet.

In no case do you have the opportunity or means to consider onPacketRecieving, and onPacketSending as being the client's ability to send/receive. You can block the server from relaying, and interacting with a client. But never directly control a client. This may seem silly to include this if you are mindful of what is actually happening, but in older versions there is implementations of ProtocolLib syntax such where, the incoming packets were wrongly and confusingly labeled client side. Everything is server side. Its just incoming packets that you change before they are processed by the rest of the server are called

Client, because its *as if* the initial request of the client was changed, but in reality the server modified the packet and then finished processing according to the added modifications.

1.4.4 Remaining censorPacket(...) Code++

If the player is the HashMap of online players who requested censorship then the process proceeds.

The message is broken apart into its text components,read, and then the JSON string is retrieved and stored (as a String). I may write more on this later, because I also experimented around this in various ways but for now I'll continue. Normally, this is where the parser, defined earlier, would be used to create a JSONObject, but I found it easier (also explained earlier) to just censor the entire String representing the JSONObject.

```
msg = msg.toLowerCase();
List<String> filter = getConfig().getStringList("filter");
for (String s : filter)
{
  if (msg.contains(s))
  {
  msg = msg.replaceAll(s, ".");
  mkNewPacket = true;//now we know we need to make a new packet
  }
}
if(mkNewPacket)
{
  WrappedChatComponent tmp = WrappedChatComponent.fromJson(msg);
  event.getPacket().getChatComponents().write(0, tmp);
}
```

The above code is among the tricker bits construct, not the filtering, but repacking the chat. Since the entire String is filtered, I can just replace the filter-words and packet the String backup, but I tried several times to create entirely new packets, and it was much more challenging, and time consuming.

I'll write something to explain the write/read functions of the ChatComponents() sometime, but I don't have time now. So another time, when I do I'll include how to better create your packets if you need to restructure more than just text, but the actual JSONObject version as well.

As a parting word of wisdom be mindful of abstract classes and null pointers when creating your own packets to replace to event packets. They don't always catch on compile, and often the plugin will even function well... until the right case is hit. That's one other reason I chose to reuse the event packet, and just modify the contents. Additional motivations being time and memory awareness. Why create new objects when the old ones will work fine?

2 The ChatPatchCommand Class