GeyserLink

GeyserLink Developers
Copyright (C) 2020 GeyserLink Developers

Table of contents

Introduction

QuickStart

How it works

Custom Messages

Contributing

Introduction

> **Note**
>
> GeyserLink is only of use to developers or plugins that rely on GeyserLink to be available. It does not itself provide any additional feature.

What is GeyserLink¶

GeyserLink aims to provide an easy method of sending messages between any server involved in a Minecraft connection and to do so in such a way as to allow both a trusted setup and an untrusted setup. This could also potentially be a useful way for client side mods to implement better communication. It is the TCP/IP of PluginMessages.

An example configuration could be a Geyser proxy connected to a Bungeecord proxy connecting to a Spigot server. If all three servers are under the control of the same user then they can be configured to trust each other. If the Geyser proxy is instead run by another user (for example someone connecting to a server using their own Proxy) then it will be untrusted but still be able to participate in communication where no trust is needed.

One example is that a trusted proxy could be queried about a players real IP whereas an untrusted one cannot be trusted to provide this and thus a plugin relying on this behaviour can gracefully fallback to using the proxy IP.

The following configurations should be supported:

- Owner who has multiple proxies connecting to them for load balancing reasons. These are trusted.
- Owner who doesn't run their own proxy but still wants to provide support for users to run their own. These are untrusted.
- Mix of the above.
- A Client Side mod connected to any of the above. In this case the client side mod would be untrusted but the servers could be trusted.

Presently GeyserLink can be used as a plugin for the following servers:

- GeyserMC
- Spigot
- Bungeecord

Copyright (C) 2020 GeyserLink Developers

Features¶

- Provide a secure messaging system utilizing the built-in minecraft plugin messages. All messages are signed with a private key and all messages can be verified by other participants as being valid.
- Automatically discovers participant keys and will record them.
- Easily convert an untrusted member into trusted by copying its public key in `dynamic.yml` config file.
- Messages are linked to their responses using a unique sequence ID.
- Provides a lambda style callback function so that message responses can be provided close in code to where messages are generated.
- Supports multiple responses as some messages may require more than one participant to respond
- Easily create custom messages

Example

Send a ping message out and write out to the log any responses received.

```
// Will get a response from every participant
GeyserLink.getInstance().sendMessage(player, new PingMessage("Hello world!"))
    .onResponse(PingResponse.class, (result, signed, response) -> {
        getLogger().info("Got a ping response: " + response);
    });
```

Example

Send a custom message and retrieve a custom complex response.

```
// Will get a response from every participant
GeyserLink.getInstance().sendMessage(player, new GetPlayerProfileMessage("bundie")
    .onResponse(GetPlayerProfileResponse.class, (result, signed, response) -> {
        // Only accept trusted responses
        if (signed.isTrusted()) {
            getLogger().info(String.format("name:%s, location:%s world:%s",
                    response.getName(), response.getLocation().toString(), respons
        }
    });
```

Last update:

Copyright (C) 2020 GeyserLink Developers

QuickStart

Maven Dependency¶

Add the following to your `pom.xml`:

```
<repositories>
    <!-- Bundabrg's Repo -->
    <repository>
        <id>bundabrg-repo</id>
        <url>https://repo.worldguard.com.au/repository/maven-public<
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>au.com.grieve.geyserlink</groupId>
        <artifactId>GeyserLink</artifactId>
        <version>1.1.0-SNAPSHOT</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Download the Plugin¶

Download and place the GeyserLink plugin into the plugins folder of each of your servers that need it.

When the server is started it will generate a `config.yml` and a `dynamic.yml` file that will contain keys for this server.

> **Note**

Copyright (C) 2020 GeyserLink Developers

GeyserMC *(https://geysermc.org)*, a bedrock to java proxy, does not yet support plugins so you will need to either wait till this pull request *(https://github.com/GeyserMC/Geyser/pull/742)* is merged or build your own version from that branch.

Sending a PingMessage¶

GeyserLink supports some build in messages that it will respond to itself. One of them is a PingMessage which simply responds with whatever data was in the PingMessage payload. This is a useful way of finding out what other GeyserLink services are available.

Example

```
GeyserLink.getInstance().sendMessage(player, new PingMessage("Hello world!"));
```

This is not terribly useful as we want to capture the response. The `sendMessage` method allows you to chain a `onResponse` call that allows you to define a lambda to run when a response to the messages is received. Note that you can get multiple responses for some messages.

Lets capture the response and print out the packet. Note that we specify what the response class is as well which in this case is a `PingResponse`.

Example

```
GeyserLink.getInstance().sendMessage(player, new PingMessage("Hello world!"))
    .onResponse(PingResponse.class, (result, signed, response) -> {
        // We have recieved a response to our ping. Print it out
        getLogger().info("Got a PingResponse: " + response);
    });
```

Last update:

Copyright (C) 2020 GeyserLink Developers

How it works

Minecraft Plugin Messages¶

GeyserLink makes use of the normal PluginMessage channel provided through a special packet in Minecraft. As a result it will not cause any issues if a server does not have GeyserLink. The plugin writer does not have to deal with Plugin Messages as they will make use of events and methods provided by GeyserLink.

GeyserLink takes a message provided by the program and will wrap it inside a `signed message` (or a `signed response`). This signed message will contain identifiers for the sequence, the sender, the packet payload and a signature for the packet. Responses are similarly wrapped and are associated to Messages by the sequence number and Sender.

The reason for the signature is that we cannot trust messages that could come from a client. This means that every message sent and received by GeyserLink has a signature affirming that the data is correct and who has sent it. GeyserLink can also be configured as to which keys that sign a signature can be trusted.

The only difference between an untrusted and trusted message is that the key used to sign a trusted message has been placed in the `trusted` field in `dynamic.yml` rather than a 'known' field. Untrusted messages can still be useful and unless there are any security implications then plugin writers should consider trying to support messages from untrusted sources or gracefully dealing with them.

> **Info**
>
> Geyserlink will send signed messages using the plugin channel `geyserlink:message` and responses using the plugin channel `geyserlink:response`. There is no need to trap these as GeyserLink provides its own events, `GeyserLinkMessageEvent` and `GeyserLinkResponseEvent` that will unwrap the message properly.

Creating a Raw Message¶

Before we create custom messages it is worth knowing how to create a raw message and response as it describes how GeyserLink works.

Here is an example:

Copyright (C) 2020 GeyserLink Developers

Example

```
GeyserLink.getInstance().sendMessage(player, "myChannel", "mySubChannel", "test".g
```

This will send a GeyserLink message with a channel `myChannel` and a sub channel of `mySubChannel`. These are similar sounding to the Plugin Message channel but are not related and are used to allow you to trap what GeyserLink messages are interesting.

GeyserLink will trigger the event `GeyserLinkMessageEvent` when a GeyserLink message is received. The following is an example of how to trap the above message in Spigot:

Example

```
@EventHandler
public void onGeyserLinkMessage(GeyserLinkMessageEvent event) {
    GeyserLinkMessage message = event.getSignedMessage().getMessage();

    if (message.getChannel().equals("myChannel") && message.getSubChannel().equals
        getLogger().info("Got a GeyserLink message with payload: " + new String(me
    }
}
```

Alert

The `channel` and `subchannel` here are not related to the Plugin Message `channel`. These are wrapped inside the message and are available for your use and it is recommended to namespace the name like the regular PluginChannel channel does to avoid collisions. GeyserLink reserves `geyserlink:main` as a channel for its own use.

Creating a Raw Response¶

Lets expand on the previous example and return something back. The Event will now look like this:

Example

Copyright (C) 2020 GeyserLink Developers

```
@EventHandler
public void onGeyserLinkMessage(GeyserLinkMessageEvent event) {
    GeyserLinkMessage message = event.getSignedMessage().getMessage();

    if (message.getChannel().equals("myChannel") && message.getSubChannel().equals
        getLogger().info("Got a GeyserLink message with payload: " + new String(me

        GeyserLink.getInstance().sendResponse(event.getPlayer(), message, "respond
    }
}
```

Note that to respond we had to pass the `message` as a parameter. This allows GeyserLink to properly link the response to the message.

Let's update how it is executed now:

Example

```
GeyserLink.getInstance().sendMessage(player, "myChannel", "mySubChannel", "test".g
    .onResponse(() -> (result, signed) -> {
        getLogger().info("Got a GeyserLink response: " + new String(signed.getMess
    });
```

Now it will output the response it receives. Note that the lambda here is passed the signed message.

Note

GeyserLink will trigger a `GeyserLinkResponseEvent` for any response but it is less useful as responses can more easily be used through the `onResponse` method as demonstrated here.

Private and Public Keys¶

When first run GeyserLink will generate a new public and private key for itself, storing these in the configuration file `dynamic.yml`. The private key should be considered confidential.

When sending signed messages GeyserLink will use its private key to sign the message. The receiving server will use the public key to verify if a signature is valid and if it is a trusted public key.

If GeyserLink receives a message by a sender it does not have the public key for it will send out a WHOIS packet asking the sender for their public key and will then add it to its known key list and persist this into the `dynamic.yml` configuration file.

Any key can be made trusted by editing the `dynamic.yml` file and moving the key into the `trusted` list.

Copyright (C) 2020 GeyserLink Developers

Last update:

Copyright (C) 2020 GeyserLink Developers

Custom Messages

Whilst we can send raw messages using GeyserLink it is easier to define a Message and Response class that will serialize and unserialize the responses.

Custom Message and Response¶

Let's create a `PlayerQueryMessage` and `PlayerQueryResponse` that returns the number of players on the server. It is up to you as to which server will respond to this but for this example we will assume the GeyserLink plugin is on a Spigot server and on a Geyser proxy and that either side may send the message to get a response from the other side.

PlayerQueryMessage¶

Example

```
@Getter
@ToString
public class PlayerQueryMessage extends WrappedMessage {
    private final String channel = "myPlugin:command";
    private final String subChannel = "player-query";

    public PlayerQueryMessage(String data) {
        super();
    }

    public PlayerQueryMessage(JsonNode node) {
        super(node);
    }

    @Override
    protected ObjectNode serialize() {
        return super.serialize();
    }
}
```

Here we define the channel and subchannel that this message will use. The rest is mainly boilerplate dealing with serializing or deserializing the object.

Info

Messages make use of JSON to contain their structure in a packet. This means that when serializing an object it will call the `serialize` method that will add to an ObjectNode any data relevant for the object.

To deserialize a constructor taking a `JsonNode` is used which then pulls from that any relevant fields for the object.

Copyright (C) 2020 GeyserLink Developers

PlayerQueryResponse¶

Example

```
@Getter
@ToString
public class PlayerQueryResponse extends WrappedResponse {
    private int count;

    public PlayerQueryResponse(int count) {
        super();

        this.count = count;
    }

    public PlayerQueryResponse(JsonNode node) {
        super(node);
        this.count = node.get("count").asInt();
    }

    @Override
    protected ObjectNode serialize() {
        return super.serialize()
                .put("count", count);
    }
}
```

This one is a bit more interesting as it has a data field `count`. We have to deal with how to deserialize from a JsonNode and how to serialize to an ObjectNode.

Note

The Response does not need to define a channel or subchannel.

Putting it together¶

Now you can send a PlayerQueryMessage by doing something like this:

Example

```
GeyserLink.getInstance().sendMessage(player, new PlayerQueryMessage())
    .onResponse(PlayerQueryResponse.class, (result, signed, response) -> {
        getLogger(String.format("The server has %d players on it", response.getCou
    });
```

Note

Copyright (C) 2020 GeyserLink Developers

When sending a raw message you only have the fields `result` and `signed`. When using a wrapped message you also get a field for the message itself. In the above case `response` will be a `PlayerQueryResponse` object and will be deserialized from the `signed` object but we still get the `signed` object as it has data on it that could be useful.

Message Event¶

Whichever server is responding to the message will need to register an event listener for the message. The following is a simple example for a Spigot server.

Example

```
@EventHandler
public void onGeyserLinkMessage(GeyserLinkMessageEvent event) {
    if (!event.getSignedMessage().getMessage().getChannel().equals("myPlugin:comma
        return;
    }

    switch(event.getSignedMessage().getMessage().getSubChannel()) {
        case "player-query":
            GeyserLink.getInstance().sendResponse(event.getPlayer(), event.getSign
                    new PlayerQueryResponse(plugin.getServer().getOnlinePlayers().
            break;
    }
}
```

For completion sake the following is for the GeyserMC server, note how similar it is.

Example

```
@Event
public void onGeyserLinkMessage(GeyserLinkMessageEvent event) {
    if (!event.getSignedMessage().getMessage().getChannel().equals("myPlugin:comma
        return;
    }

    switch(event.getSignedMessage().getMessage().getSubChannel()) {
        case "player-query":
            GeyserLink.getInstance().sendResponse(event.getSession(), event.getSig
                    new PlayerQueryResponse(plugin.getConnector().getPlayers().val
            break;
    }
}
```

Copyright (C) 2020 GeyserLink Developers

Security¶

You may have noticed an issue with the previous example. Anyone could connect to a server and either spoof GeyserLink messages or run their own GeyserLink plugin on a proxy and thus the player count must come from a trusted partner. We also don't want a random person being able to query the player count.

Note

If possible try support untrusted clients as well. If the player count in this example is not confidential then there may be no reason not to still allow queries. Also if the message is coming from a client side mod then it will be untrusted by default.

To solve this we need to check on both sides. On the client side the following would only accept trusted responses:

Example

```
GeyserLink.getInstance().sendMessage(player, new PlayerQueryMessage())
    .onResponse(PlayerQueryResponse.class, (result, signed, response) -> {
        if (signed.isTrusted()) {
            getLogger(String.format("The server has %d players on it", response.ge
        }
    });
```

The server could be updated as follows:

Example

```
@EventHandler
public void onGeyserLinkMessage(GeyserLinkMessageEvent event) {
    if (!event.getSignedMessage().getMessage().getChannel().equals("myPlugin:comma
        return;
    }

    switch(event.getSignedMessage().getMessage().getSubChannel()) {
        case "player-query":
            if (event.getSignedMessage().isTrusted()) {
                GeyserLink.getInstance().sendResponse(event.getPlayer(), event.get
                        new PlayerQueryResponse(plugin.getServer().getOnlinePlayer
            }
            break;
    }
}
```

Last update:

Copyright (C) 2020 GeyserLink Developers

Contributing¶

Here are some ways that you can help contribute to this project.

New ideas or Bug Reports¶

Need something? Found a bug? Or just have a brilliant idea? Head to the Issues *(https://github.com/Bundabrg/GeyserLink/issues)* and create new one.

Contributing Code¶

If you know Java then take a look at open issues and create a pull request.

Do the following to build the code:

```
git clone https://github.com/Bundabrg/GeyserLink
cd GeyserLink
mvn clean package
```

Contributing Documentation¶

If you can help improve the documentation it would be highly appreciated. Have a look under the `docs` folder for the existing documentation.

The documentation is built using `mkdocs`. You can set up a hot-build dev environment that will auto-refresh changes as they are made.

Requirements¶

- python3
- pip3
- npm (only if changing themes)

Install dependencies by running:

```
pip3 install -r requirements.txt
```

Copyright (C) 2020 GeyserLink Developers

Dev Environment¶

To start a http document server on `http://127.0.0.1:8000` execute:

```
mkdocs serve
```

Change PDF Theme¶

Edit the PDF theme under `docs/theme/pdf`. Rebuild by doing the following:

```
cd docs/theme/pdf
npm install
npm run build-compressed
```

This will update `pdf.css` under `docs/css/pdf.css`. Rebuilding the docs will now use the new theme.

Last update:

Copyright (C) 2020 GeyserLink Developers