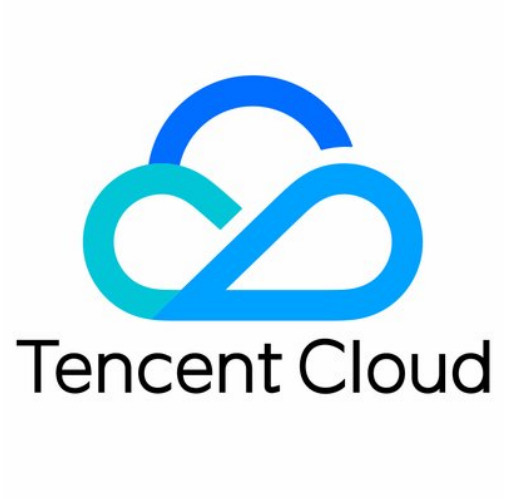
Game Server Elastic-scaling

Development Guide

Product Documentation



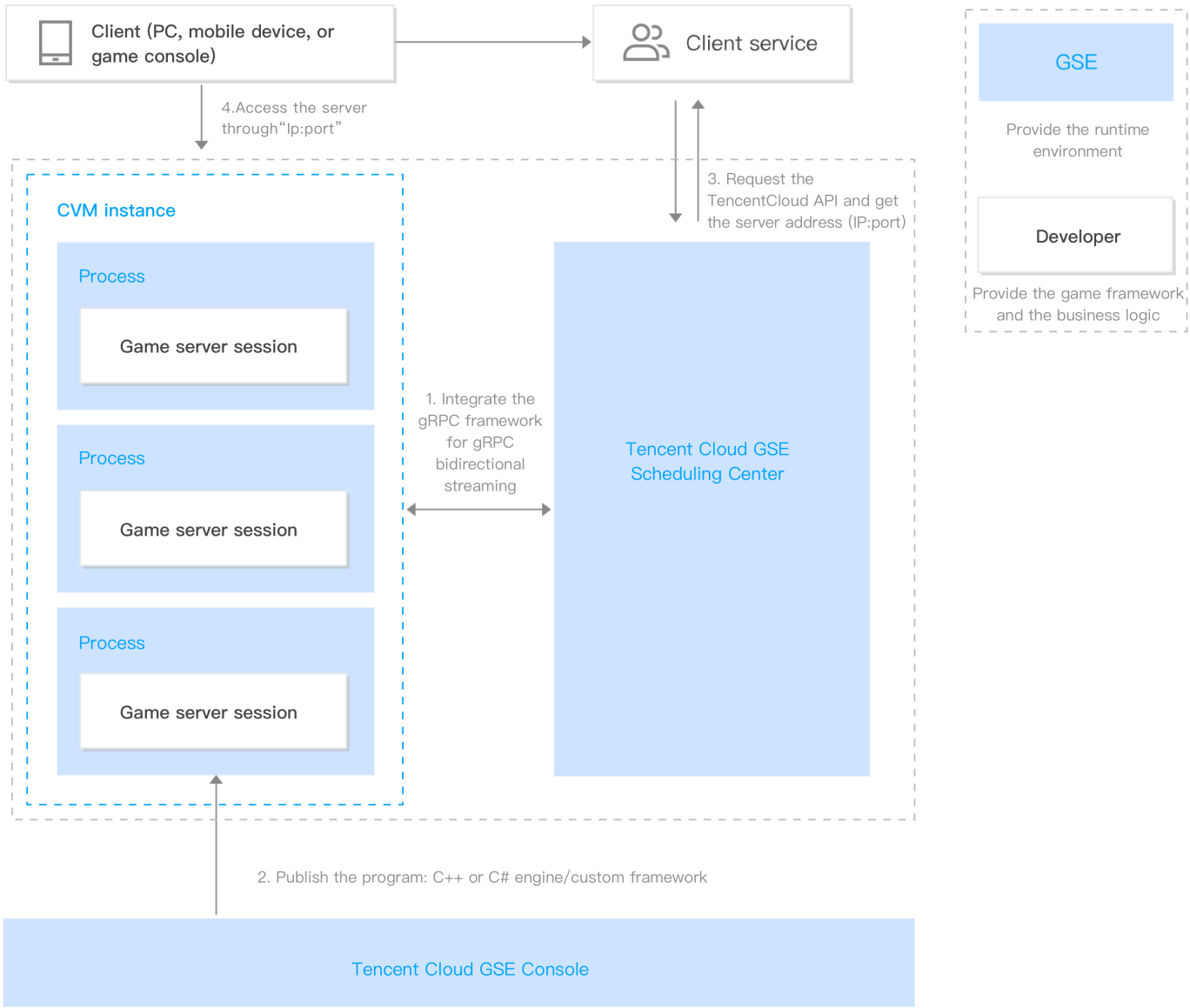
Contents

Development Guide
- Overall Process
- Integrating Server with gRPC Framework
  - gRPC C++ Tutorial
  - gRPC C# Tutorial
  - gRPC Go Tutorial
  - gRPC Java Tutorial
  - gRPC Lua Tutorial
  - gRPC Node.js Tutorial
  - gRPC Unity Tutorial
- Getting Server Address
  - TencentCloud API Calling Method
  - Creating Game Server Session
  - Placing Game Server Session
- GSE Local
- Latency Test Tool
- Game Process Launch Configuration

Development Guide
Overall Process

Last updated : 2020-09-08 15:27:29

Overall Flowchart



Integration Steps

Step 1. Integrate the server with the gRPC framework

The game server communicates with GSE over gRPC. The gRPC framework can be integrated with the game server program in various programming languages to generate game server executable files. For more information on how to integrate GSE with the server in different languages, please see gRPC - C++ Tutorial, gRPC - C# Tutorial, gRPC - Go Tutorial, gRPC - Java Tutorial, gRPC - Lua Tutorial, and gRPC - Node.js Tutorial. For other languages, please see the gRPC official documentation.

Step 2. Publish the program

1. Upload an asset package
   An asset package contains the executable files, dependencies, and installation script of the game server. You need to package them as a ZIP file before upload. For more information, please see Creating Code Packages.
2. Create a server fleet
   Deploy the uploaded asset package on the created server fleet and complete process management, deployment configuration, scaling configuration, etc. For more information, please see Creating Server Fleets.

Step 3. Call a TencentCloud API to get the server address (IP:port)

You can get the server address (IP:port) by creating or placing a game server session.

Method 1. Create a game server session

Call a TencentCloud API:
The client TencentCloud API call process varies by supporting mode of the game server session.

- When a game server session only supports one game:
  - Create a game server session (CreateGameServerSession);
  - Join a game server session (JoinGameServerSession).
- When a game server session supports multiple games or one service (such as login):
  - Query the game server session list (DescribeGameServerSessions) or search in the game server session list (SearchGameServerSessions);
  - If there is a game server session, join it (JoinGameServerSession);
  - If there is no game server session, create one (CreateGameServerSession) and join it (JoinGameServerSession).

For more information on how to call TencentCloud APIs, please see Creating Game Server Session.

Method 2. Place a game server session

- Call a TencentCloud API:

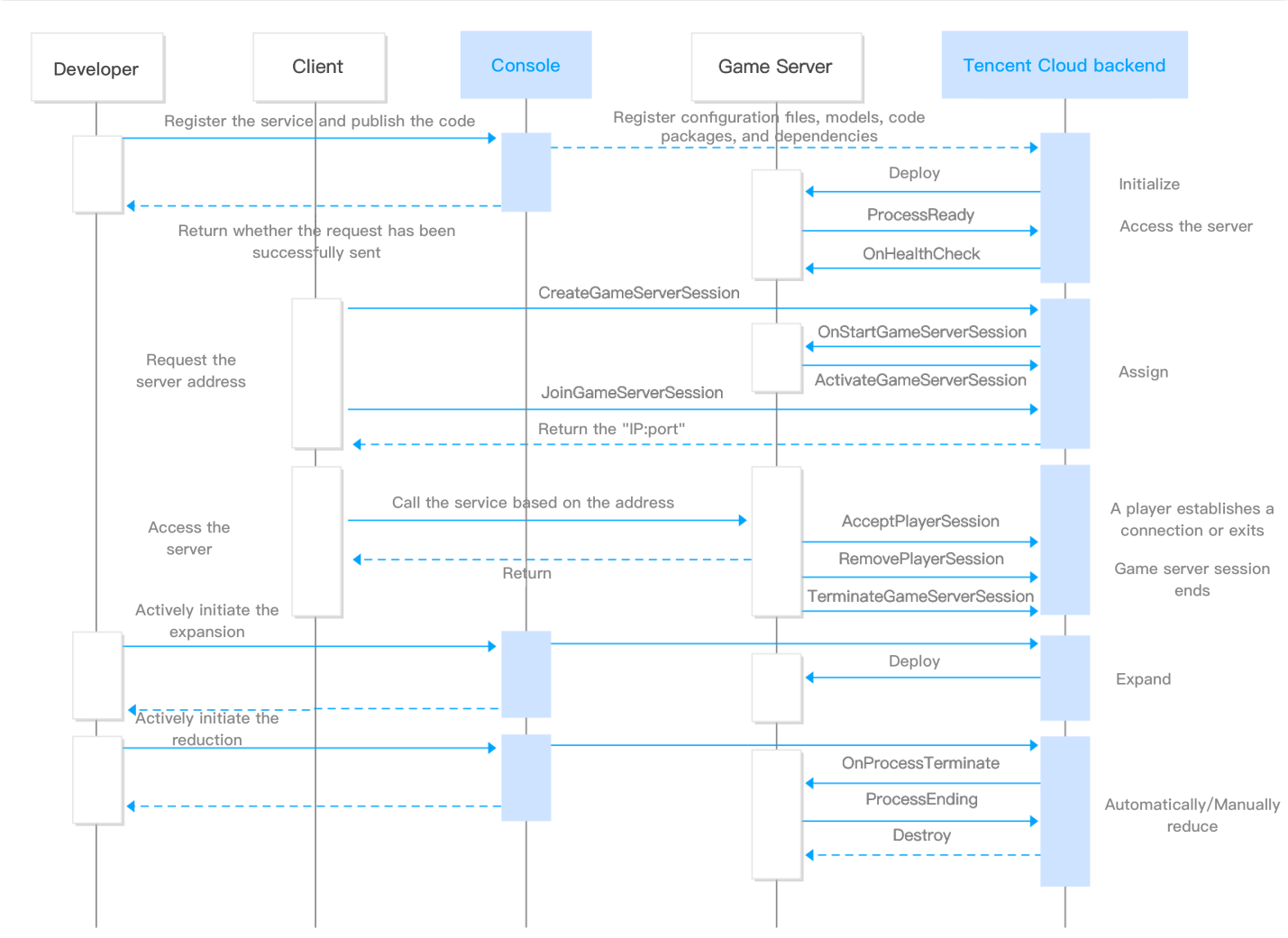
- Start placing a game server session (StartGameServerSessionPlacement);
- Query game server session placement (DescribeGameServerSessionPlacement);
- Stop placing a game server session (StopGameServerSessionPlacement).

For more information on how to call the TencentCloud APIs, please see Placing Game Server Session.

Step 4. The client uses the IP:port to access the server

The client can connect to the target server through the `IP:port` returned in step 3.

Workflow



Integrating Server with gRPC Framework
gRPC C++ Tutorial

Last updated : 2021-11-17 18:06:09

Installing gRPC

1. Prerequisites: install CMake.
   - Linux

     ```
     $ sudo apt install -y cmake
     ```

   - MAC OS

     ```
     $ brew install cmake
     ```

2. Install gRPC and Protocol Buffers locally.

> Note
> For more information on the installation process, see Installing CMake, Installing gRPC C++, and Installing Protocol Buffers.

Defining Service

gRPC uses Protocol Buffers to define a service: an RPC service specifies methods that can be called remotely by using parameters and return types.

> Note
> We provide the .proto files for service definition. You can directly download them with no need to generate them by yourself.

Generating gRPC Code

1. After defining the service, you can use protoc (protocol buffer compiler) to generate the client and server code (in any language supported by gRPC).
2. The generated code includes the client stub and the abstract APIs to be implemented by the server.
3. Steps for generating gRPC code:
   In the `proto` directory, run:

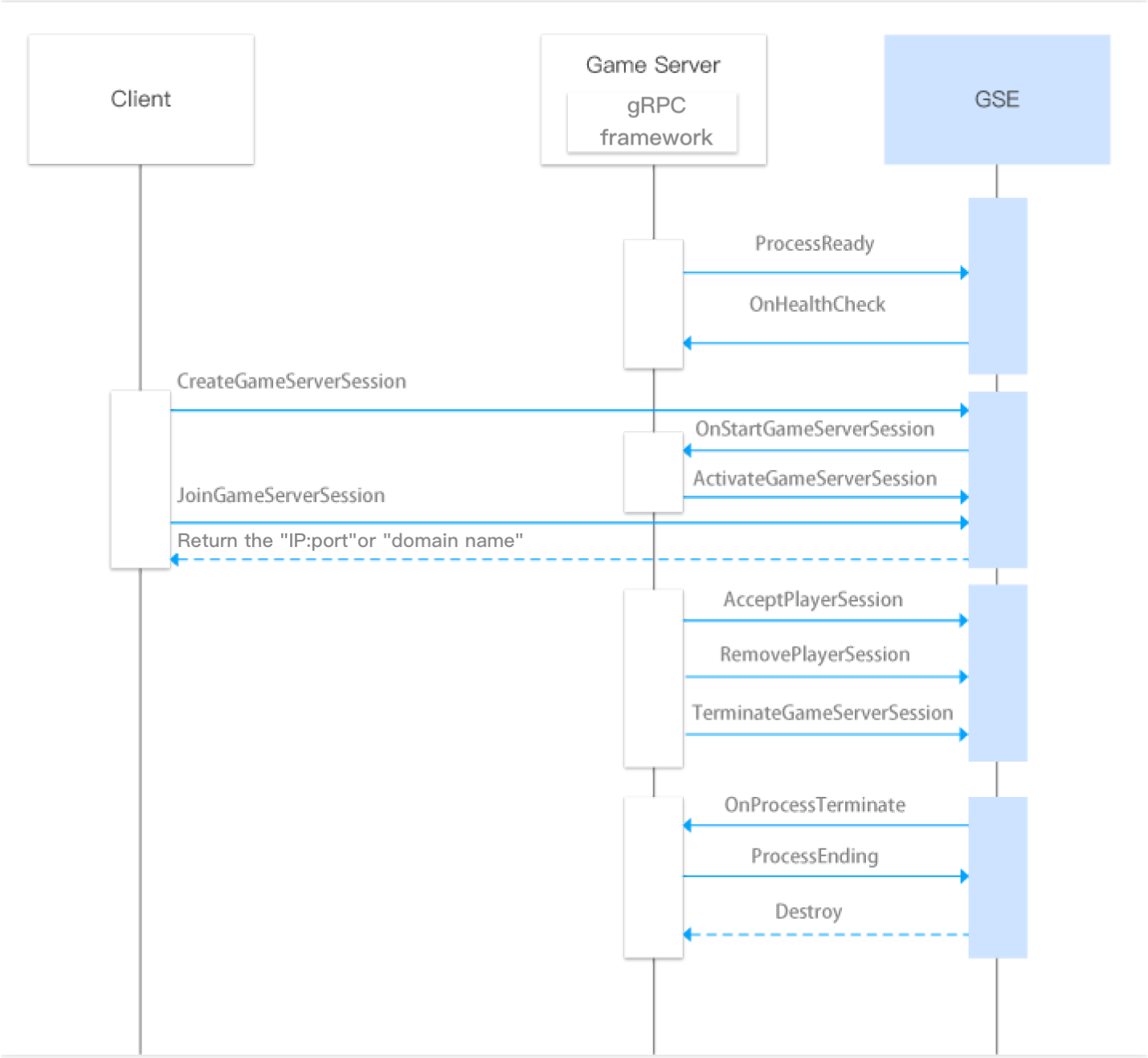
   ```
   protoc --cpp_out=. *.proto
   ```

   to generate the `pb.cc` and `pb.h` files.

```
protoc --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` *.proto
```

to generate the corresponding gRPC code.
Move the eight generated files to an appropriate location in the project.

Game Process Integration Process



Game server callback API list

API Name	API Description
OnHealthCheck	Runs health check
OnStartGameServerSession	Receives game server session
OnProcessTerminate	Ends game process

Game server active API list

API Name	API Description
ProcessReady	Gets process ready
ActivateGameServerSession	Activates game server session
AcceptPlayerSession	Receives player session
RemovePlayerSession	Removes player session
DescribePlayerSessions	Gets player session list
UpdatePlayerSessionCreationPolicy	Updates player session creation policy
TerminateGameServerSession	Ends game server session
ProcessEnding	Ends process
ReportCustomData	Reports custom data

Others

When the game process uses gRPC to call a game server active API, you need to add two fields to `meta` of the gRPC request.

Field	Description	Type
pid	`pid` of the current game process	string
requestId	`requestId` of the current request, which is used to uniquely identify a request	string

1. Generally, after the server is initialized, the process will check itself to see whether it can provide services, and the game server will call the `ProcessReady` API to notify GSE that the process is ready to host a game server session. After receiving the notification, GSE will change the status of the server instance to "Active".

```
Status GseManager::ProcessReady(std::vector<std::string> &amp;logPath, int clientPort, int grp
cPort, GseResponse&amp; reply)
{
ProcessReadyRequest request;
// Log path
for (auto iter = logPath.begin(); iter != logPath.end(); iter++)
{
request.add_logpathstoupload(*iter);
}
```

```
GConsoleLog-&gt;PrintOut(true, "ProcessReady clientPort is %d¥n", clientPort);
GConsoleLog-&gt;PrintOut(true, "ProcessReady grpcPort is %d¥n", grpcPort);

// Set the ports
request.set_clientport(clientPort);
request.set_grpcport(grpcPort);

ClientContext context;
AddMetadata(context);

// Ready to provide services
return stub_-&gt;ProcessReady(&amp;context, request, &amp;reply);
}
```

2. After the process is ready, GSE will call the `OnHealthCheck` API to perform a health check on the game server every minute. If the health check fails three consecutive times, the process will be considered to be unhealthy, and no game server sessions will be assigned to it.

```
Status GameServerGrpcSdkServiceImpl::OnHealthCheck(ServerContext* context, const HealthCheckRe
quest* request, HealthCheckResponse* reply)
{
reply->set_healthstatus(healthStatus);
return Status::OK;
}
```

3. Because the client calls the CreateGameServerSession API to create a game server session and assigns it to a process, GSE will be triggered to call the `onStartGameServerSession` API for the process and change the status of `GameServerSession` to "Activating".

```
Status GameServerGrpcSdkServiceImpl::OnStartGameServerSession(ServerContext* context, const St
artGameServerSessionRequest* request, GseResponse* reply)
{
auto gameServerSession = request->gameserversession();
GGseManager->SetGameServerSession(gameServerSession);
GseResponse processReadyReply;
Status status = GGseManager->ActivateGameServerSession(gameServerSession.gameserversessionid()
, gameServerSession.maxplayers(), processReadyReply);
// Determine whether the activation has succeeded based on `status` and `replay`
```

```
return Status::OK;
}
```

4. After the game server receives `onStartGameServerSession`, you need to handle the logic or resource allocation by yourself. After everything is ready, the game server will call the `ActivateGameServerSession` API to notify GSE that the game server session has been assigned to a process and is ready to receive player requests and will change the server status to "Active".

```
Status GseManager::ActivateGameServerSession(std::string gameServerSessionId, int maxPlayers, Gse
Response& reply)
{
GConsoleLog->PrintOut(true, "ActivateGameServerSession gameServerSessionId is %s¥n", gameServerSe
ssionId.c_str());
GConsoleLog->PrintOut(true, "ActivateGameServerSession maxPlayers is %d¥n", maxPlayers);
ActivateGameServerSessionRequest request;
request.set_gameserversessionid(gameServerSessionId);
request.set_maxplayers(maxPlayers);

ClientContext context;
AddMetadata(context);

return stub_->ActivateGameServerSession(&context, request, &reply);
}
```

5. After the client calls the JoinGameServerSession API for the player to join, the game server will call the `AcceptPlayerSession` API to verify the validity of the player. If the connection is accepted, the status of `PlayerSession` will be set to "Active". If the client receives no response within 60 seconds after calling the `JoinGameServerSession` API, it will change the status of `PlayerSession` to "Timeout" and then call `JoinGameServerSession` again.

```
Status GseManager::AcceptPlayerSession(std::string playerSessionId, GseResponse& reply)
{
AcceptPlayerSessionRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
request.set_playersessionid(playerSessionId);
ClientContext context;
AddMetadata(context);
```

```
return stub_->AcceptPlayerSession(&context, request, &reply);
}
```

6. After the game ends or the player exits, the game server will call the `RemovePlayerSession` API to remove the player, change the status of `playersession` to "Completed", and reserve the player slot in the game server session.

```
Status GseManager::RemovePlayerSession(std::string playerSessionId, GseResponse& reply)
{
GConsoleLog->PrintOut(true, "RemovePlayerSession playerSessionId is %s¥n", playerSessionId.c_str
());
RemovePlayerSessionRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
request.set_playersessionid(playerSessionId);
ClientContext context;
AddMetadata(context);

return stub_->RemovePlayerSession(&context, request, &reply);
}
```

7. After a game server session (a game battle or a service) ends, the game server will call the `TerminateGameServerSession` API to end the `GameServerSession` and change its status to `Terminated`.

```
Status GseManager::TerminateGameServerSession(GseResponse& reply)
{
GConsoleLog->PrintOut(true, "start to TerminateGameServerSession¥n");
TerminateGameServerSessionRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
ClientContext context;
AddMetadata(context);


return stub_->TerminateGameServerSession(&context, request, &reply);
}
```

8. In case of health check failure or reduction, GSE will call the `OnProcessTerminate` API to end the game process. The reduction will be triggered according to the protection policy configured in the GSE console.

```
Status GameServerGrpcSdkServiceImpl::OnProcessTerminate(ServerContext* context, const ProcessT
erminateRequest* request, GseResponse* reply)
{
auto terminationTime = request->terminationtime();
GGseManager->SetTerminationTime(terminationTime);
// If the following two APIs are called, the game server session will be ended immediately. We
recommend you call `ProcessEnding` to end the process only when there are no players or game s
erver sessions
// If the following two APIs are not called, `ProcessEnding` will be called to end the process
according to the protection policy. We recommend you configure time-period protection


//End the game server sessions
GseResponse terminateGameServerSessionReply;
GGseManager->TerminateGameServerSession(terminateGameServerSessionReply);

// End the processes
GseResponse processEndingReply;
GGseManager->ProcessEnding(processEndingReply);

return Status::OK;
}
```

9. The game server calls the `ProcessEnding` API to end the process immediately, change the server process status to "Terminated", and repossess the resources.

```
// Active call: a game battle corresponds to a process. The `ProcessEnding` API will be active
ly called after the game battle ends
// Passive call: in case of reduction, process exception, or health check failure, the `Proces
sEnding` API will be called passively according to the protection policy. If a full protection
or time-period protection policy is configured, it is required to determine whether there are
any players in the game server session before the passive call can be made
Status GseManager::ProcessEnding(GseResponse& reply)
{
GConsoleLog->PrintOut(true, ”start to ProcessEnding¥n”);
ProcessEndingRequest request;
ClientContext context;
AddMetadata(context);
```

```
return stub_->ProcessEnding(&context, request, &reply);
}
```

10. The game server calls the `DescribePlayerSessions` API to get the information of the player in the game server session (which is optional based on your actual business needs).

```
Status GseManager::DescribePlayerSessions(std::string gameServerSessionId, std::string playerId,
std::string playerSessionId,std::string playerSessionStatusFilter, std::string nextToken, int lim
it, DescribePlayerSessionsResponse& reply)
{
GConsoleLog->PrintOut(true, "start to DescribePlayerSessions¥n");
DescribePlayerSessionsRequest request;
request.set_gameserversessionid(gameServerSessionId);
request.set_playerid(playerId);
request.set_playersessionid(playerSessionId);
request.set_playersessionstatusfilter(playerSessionStatusFilter);
request.set_nexttoken(nextToken);
request.set_limit(limit);
ClientContext context;
AddMetadata(context);

return stub_->DescribePlayerSessions(&context, request, &reply);
}
```

1. The game server calls the `UpdatePlayerSessionCreationPolicy` API to update the player session creation policy and set whether to accept new players, i.e., whether to allow new players to join a game session (which is optional based on your actual business needs).

```
Status GseManager::UpdatePlayerSessionCreationPolicy(std::string newpolicy, GseResponse& reply
)
{
GConsoleLog->PrintOut(true, "UpdatePlayerSessionCreationPolicy, newpolicy is %s¥n", newpolicy.
c_str());
UpdatePlayerSessionCreationPolicyRequest request;
request.set_gameserversessionid(gameServerSession.gameserversessionid());
request.set_newplayersessioncreationpolicy(newpolicy);
ClientContext context;
AddMetadata(context);
```

```
return stub_->UpdatePlayerSessionCreationPolicy(&context, request, &reply);
}
```

12. The game server calls the `ReportCustomData` API to notify GSE of the custom data that can be viewed during game server session query (which is optional based on your actual business needs).

```
Status GseManager::ReportCustomData(int currentCustomCount, int maxCustomCount, GseResponse& repl
y)
{
GConsoleLog->PrintOut(true, "ReportCustomData, currentCustomCount is %d¥n", currentCustomCount);
GConsoleLog->PrintOut(true, "ReportCustomData, maxCustomCount is %d¥n", maxCustomCount);
ReportCustomDataRequest request;
request.set_currentcustomcount(currentCustomCount);
request.set_maxcustomcount(maxCustomCount);

ClientContext context;
AddMetadata(context);

return stub_->ReportCustomData(&context, request, &reply);
}
```

Launching Server for GSE to Call

Server running: launch `GrpcServer` .

```
GameServerGrpcSdkServiceImpl::GameServerGrpcSdkServiceImpl() : serverAddress("127.0.0.1:0"), heal
thStatus(true)
{
sem_init(&sem, 0, 0);
}
void GameServerGrpcSdkServiceImpl::StartGrpcServer()
{
ServerBuilder builder;
builder.AddListeningPort(serverAddress, grpc::InsecureServerCredentials(), &grpcPort);
builder.RegisterService(this);
std::unique_ptr<Server> server(builder.BuildAndStart());
sem_post(&sem);
server->Wait();
}
```

Connecting Client to gRPC Server of GSE

Server connecting: create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.

```
void GseManager::InitStub()
{
auto channel = grpc::CreateChannel("127.0.0.1:5758", grpc::InsecureChannelCredentials());
stub_ = GseGrpcSdkService::NewStub(channel);
}
```

Demo for C++

1. Click here to download the code of the Demo for C++.
2. Generate the gRPC code.
   As the gRPC code has already been generated in the `cpp-demo/source/grpcsdk` directory of the Demo for C++, you do not need to generate it again.
3. Launch the server for GSE to call.
   - Implement the server.
     `grpcserver.cpp` in the `cpp-demo/source/api` directory implements three server APIs.
   - Run the server.
     `grpcserver.cpp` in the `cpp-demo/source/api` directory launches `GrpcServer` .
4. Connect the client to the gRPC server of GSE.
   - Implement the client.
     `gsemanager.cpp` in the `cpp-demo/source/gsemanager` directory implements nine client APIs.
   - Connect to the server.
     Create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.
5. Compile and run the project.
   i. Install CMake.
   ii. Install GCC v4.9 or above.
   iii. Download the code and run the following command in the `cpp-demo` directory:

   ```
   mkdir build
   cmake ..
   make
   ```

   The corresponding `cpp-demo` executable file will be generated.

iv. Package the `cpp-demo` executable file as an asset package and configure the launch path as `cpp-demo` with no launch parameter needed.
v. Create a server fleet and deploy the asset package on it. After that, you can perform various operations such as scaling.

gRPC C# Tutorial

Last updated : 2021-11-17 18:06:09

Installing gRPC

1. To use gRPC C#, you need to install .Net Core 3.1 SDK first. Taking CentOS as an example, the version must be v7, v8 or above.
   - Add the signature key

     ```
     sudo rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm
     ```

   - Install .NET Core SDK

     ```
     sudo yum install dotnet-sdk-3.1
     ```

2. In addition, you can also use gRPC C# in the following runtime environments/IDEs:
   - Windows: .NET Framework 4.5 or higher, Visual Studio 2013 or higher, Visual Studio Code.
   - Linux: Mono 4 or higher, Visual Studio Code.
   - macOS X: Mono 4 or higher, Visual Studio Code, Visual Studio for Mac.

> Note
> For more information on the installation process, please see Installing gRPC C#.

Defining Service

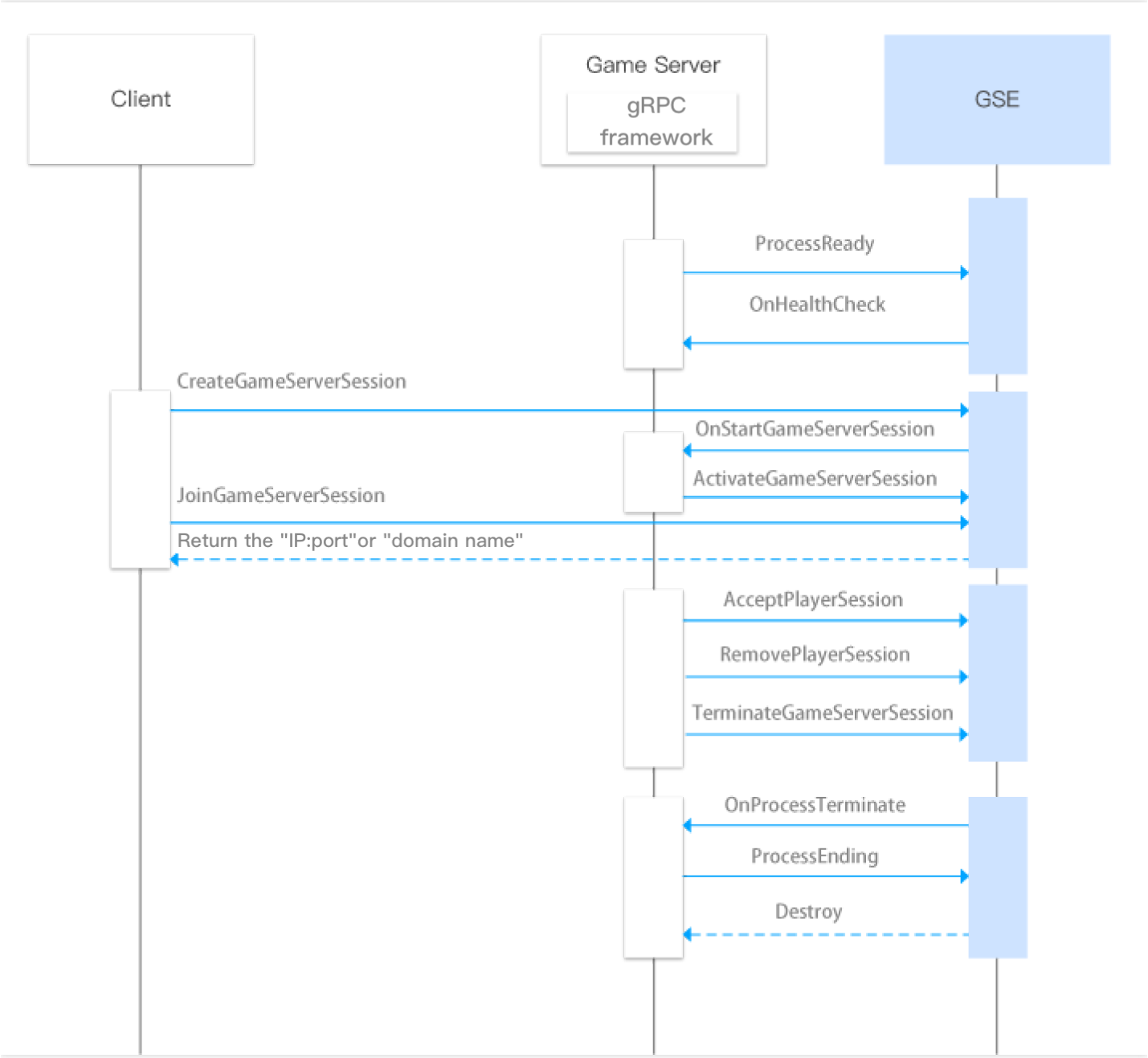
gRPC uses Protocol Buffers to define a service: an RPC service specifies methods that can be called remotely by using parameters and return types.

> Note :
> We provide the .proto files for service definition. You can click here to directly download them with no need to generate them by yourself.

Generating gRPC Code

1. After defining the service, you can use protoc (protocol buffer compiler) to generate the client and server code (in any language supported by gRPC).
2. The generated code includes the client stub and the abstract APIs to be implemented by the server.
3. Steps for generating gRPC code:
   - Download the code. In the `csharp-demo` directory, run
     ```
     dotnet run
     ```
     to automatically compile and run the service.
   - After the program is compiled and run correctly, the project's dependent libraries and binary files, and the .cs files created by compiling the `proto` file will be generated in the `csharp-demo/obj/Debug/netcoreapp3.1` folder.
   - The `proto` file is imported in `csharp-demo/csharpdemo.csproj` :
     ```
     <Protobuf Include="..¥proto¥csharp-demo¥GameServerGrpcSdkService.proto" Link="GameServerGrpcSdkService.proto"/>
     <Protobuf Include="..¥proto¥csharp-demo¥GseGrpcSdkService.proto" Link="GseGrpcSdkService.proto" />
     ```
     The project relies on the two `proto` files `GameServerGrpcSdkService.proto` and `GseGrpcSdkService.proto` in the `proto/csharp-demo` folder.

Game Process Integration Process



Game server callback API list

API Name	API Description
OnHealthCheck	Runs health check
OnStartGameServerSession	Receives game server session
OnProcessTerminate	Ends game process

Game server active API list

API Name	API Description
ProcessReady	Gets process ready
ActivateGameServerSession	Activates game server session
AcceptPlayerSession	Receives player session
RemovePlayerSession	Removes player session
DescribePlayerSessions	Gets player session list
UpdatePlayerSessionCreationPolicy	Updates player session creation policy
TerminateGameServerSession	Ends game server session
ProcessEnding	Ends process
ReportCustomData	Reports custom data

Others

When the game process uses gRPC to call a game server active API, you need to add two fields to `meta` of the gRPC request.

Field	Description	Type
pid	`pid` of the current game process	string
requestId	`requestId` of the current request, which is used to uniquely identify a request	string

1. Generally, after the server is initialized, the process will check itself to see whether it can provide services, and the game server will call the `ProcessReady` API to notify GSE that the process is ready to host a game server session. After receiving the notification, GSE will change the status of the server instance to "Active".

```
public static GseResponse ProcessReady(string[] logPath, int clientPort, int grpcPort)
{
logger.Println($"Getting process ready, LogPath: {logPath}, ClientPort: {clientPort}, GrpcPor
t: {grpcPort}");
// Set the ports
var req = new ProcessReadyRequest{
ClientPort = clientPort,
GrpcPort = grpcPort,
};
```

```
// Log path
req.LogPathsToUpload.Add(logPath); // After being parsed by `pb`, the `repeated` type is read-only and needs to be added by running `Add`
// Ready to provide services
return GrpcClient.GseClient.ProcessReady(req, meta);
}
```

2. After the process is ready, GSE will call the `OnHealthCheck` API to perform a health check on the game server every minute. If the health check fails three consecutive times, the process will be considered to be unhealthy, and no game server sessions will be assigned to it.

```
public override Task<HealthCheckResponse> OnHealthCheck(HealthCheckRequest request, ServerCallContext context)
{
logger.Println($"OnHealthCheck, HealthStatus: {GseManager.HealthStatus}");
return Task.FromResult(new HealthCheckResponse{
HealthStatus = GseManager.HealthStatus
});
}
```

3. Because the client calls the CreateGameServerSession API to create a game server session and assigns it to a process, GSE will be triggered to call the `onStartGameServerSession` API for the process and change the status of `GameServerSession` to "Activating".

```
public override Task<GseResponse> OnStartGameServerSession(StartGameServerSessionRequest request, ServerCallContext context)
{
logger.Println($"OnStartGameServerSession, request: {request}");
GseManager.SetGameServerSession(request.GameServerSession);
var resp = GseManager.ActivateGameServerSession(request.GameServerSession.GameServerSessionId, request.GameServerSession.MaxPlayers);
return Task.FromResult(resp);
}
```

4. After the game server receives `onStartGameServerSession`, you need to handle the logic or resource allocation by yourself. After everything is ready, the game server will call the `ActivateGameServerSession` API to notify GSE that the game server session has been assigned to a process and is ready to receive player requests and will change the server status to "Active".

```
public static GseResponse ActivateGameServerSession(string gameServerSessionId, int maxPlayers)
```

```
{
logger.Println($"Activating game server session, GameServerSessionId: {gameServerSessionId}, M
axPlayers: {maxPlayers}");
var req = new ActivateGameServerSessionRequest{
GameServerSessionId = gameServerSessionId,
MaxPlayers = maxPlayers,
};
return GrpcClient.GseClient.ActivateGameServerSession(req, meta);
}
```

5. After the client calls the JoinGameServerSession API for the player to join, the game server will call the `AcceptPlayerSession` API to verify the validity of the player. If the connection is accepted, the status of `PlayerSession` will be set to "Active". If the client receives no response within 60 seconds after calling the `JoinGameServerSession` API, it will change the status of `PlayerSession` to "Timeout" and then call `JoinGameServerSession` again.

```
public static GseResponse AcceptPlayerSession(string playerSessionId)
{
logger.Println($"Accepting player session, PlayerSessionId: {playerSessionId}");
var req = new AcceptPlayerSessionRequest{
GameServerSessionId = gameServerSession.GameServerSessionId,
PlayerSessionId = playerSessionId,
};
return GrpcClient.GseClient.AcceptPlayerSession(req, meta);
}
```

6. After the game ends or the player exits, the game server will call the `RemovePlayerSession` API to remove the player, change the status of `playersession` to "Completed", and reserve the player slot in the game server session.

```
public static GseResponse RemovePlayerSession(string playerSessionId)
{
logger.Println($"Removing player session, PlayerSessionId: {playerSessionId}");
var req = new RemovePlayerSessionRequest{
GameServerSessionId = gameServerSession.GameServerSessionId,
PlayerSessionId = playerSessionId,
};
return GrpcClient.GseClient.RemovePlayerSession(req, meta);
}
```

7. After a game server session (a game battle or a service) ends, the game server will call the `TerminateGameServerSession` API to end the `GameServerSession` and change its status to `Terminated`.

```
public static GseResponse TerminateGameServerSession()
{
logger.Println($"Terminating game server session, GameServerSessionId: {gameServerSession.GameServerSessionId}");
var req = new TerminateGameServerSessionRequest{
GameServerSessionId = gameServerSession.GameServerSessionId
};
return GrpcClient.GseClient.TerminateGameServerSession(req, meta);
}
```

8. In case of health check failure or reduction, GSE will call the `OnProcessTerminate` API to end the game process. The reduction will be triggered according to the protection policy configured in the GSE console.

```
public override Task<gseresponse> OnProcessTerminate(ProcessTerminateRequest request, ServerCallContext context)
{
logger.Println($"OnProcessTerminate, request: {request}");
// Set the process termination time
GseManager.SetTerminationTime(request.TerminationTime);
// If the following two APIs are called, the game server session will be ended immediately. We recommend you call `ProcessEnding` to end the process only when there are no players or game server sessions
// If the following two APIs are not called, `ProcessEnding` will be called to end the process according to the protection policy. We recommend you configure time-period protection
// Terminate game server sessions
GseManager.TerminateGameServerSession();
// Exit the process
GseManager.ProcessEnding();
return Task.FromResult(new GseResponse{
Status = GseResponse.Types.Status.Ok,
ResponseData = "SUCCESS",
});
}
```

9. The game server calls the `ProcessEnding` API to end the process immediately, change the server process status to "Terminated", and repossess the resources.

```
// Active call: a game battle corresponds to a process. The `ProcessEnding` API will be actively called after the game battle ends
// Passive call: in case of reduction, process exception, or health check failure, the `ProcessEnding` API will be called passively according to the protection policy. If a full protection or time-period protection policy is configured, it is required to determine whether there are any players in the game server session before the passive call can be made
public static GseResponse ProcessEnding()
{
logger.Println($"Process ending, pid: {pid}");
var req = new ProcessEndingRequest();
return GrpcClient.GseClient.ProcessEnding(req, meta);
}
```

0. The game server calls the `DescribePlayerSessions` API to get the information of the player in the game server session (which is optional based on your actual business needs).

```
public static DescribePlayerSessionsResponse DescribePlayerSessions(string gameServerSessionId, string playerId, string playerSessionId, string playerSessionStatusFilter, string nextToken, int limit)
{
logger.Println($"Describing player session, GameServerSessionId: {gameServerSessionId}, PlayerId: {playerId}, PlayerSessionId: {playerSessionId}, PlayerSessionStatusFilter: {playerSessionStatusFilter}, NextToken: {nextToken}, Limit: {limit}");
var req = new DescribePlayerSessionsRequest{
GameServerSessionId = gameServerSessionId,
PlayerId = playerId,
PlayerSessionId = playerSessionId,
PlayerSessionStatusFilter = playerSessionStatusFilter,
NextToken = nextToken,
Limit = limit,
};
return GrpcClient.GseClient.DescribePlayerSessions(req, meta);
}
```

1. The game server calls the `UpdatePlayerSessionCreationPolicy` API to update the player session creation policy and set whether to accept new players, i.e., whether to allow new players to join a game session (which is optional based on your actual business needs).

```
public static GseResponse UpdatePlayerSessionCreationPolicy(string newPolicy)
{
logger.Println($"Updating player session creation policy, newPolicy: {newPolicy}");
var req = new UpdatePlayerSessionCreationPolicyRequest{
GameServerSessionId = gameServerSession.GameServerSessionId,
```

```
NewPlayerSessionCreationPolicy = newPolicy,
};
return GrpcClient.GseClient.UpdatePlayerSessionCreationPolicy(req, meta);
}
```

2. The game server calls the `ReportCustomData` API to notify GSE of the custom data (which is optional based on your actual business needs).

```
public static GseResponse ReportCustomData(int currentCustomCount, int maxCustomCount)
{
logger.Println($"Reporting custom data, CurrentCustomCount: {currentCustomCount}, MaxCustomCou
nt: {maxCustomCount}");
var req = new ReportCustomDataRequest{
CurrentCustomCount = currentCustomCount,
MaxCustomCount = maxCustomCount,
};
return GrpcClient.GseClient.ReportCustomData(req, meta);
}
```

Launching Server for GSE to Call

Server running: launch `GrpcServer` .

```
public class Program
{
public static int ClientPort = PortServer.GenerateRandomPort(2000, 6000);
public static int GrpcPort = PortServer.GenerateRandomPort(6001, 10000);

public static void Main(string[] args)
{
CreateHostBuilder(args).Build().Run();
}
public static IHostBuilder CreateHostBuilder(string[] args) =&gt;
Host.CreateDefaultBuilder(args)
.ConfigureWebHostDefaults(webBuilder =&gt;
{
webBuilder.ConfigureKestrel(options =&gt;
{
// gRPC port (set the HTTP/2 endpoint without TLS certificate)
options.ListenAnyIP(GrpcPort, o =&gt; o.Protocols =
HttpProtocols.Http2);

// HTTP port
```

```
options.ListenAnyIP(ClientPort);
});

webBuilder.UseStartup<startup>();
});
}
```

Connecting Client to gRPC Server of GSE

Server connecting: create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.

```
public class GrpcClient
{
private static string agentAdress = "127.0.0.1:5758";
public static GameServerGrpcSdkService.GameServerGrpcSdkServiceClient GameServerClient
{
get
{
Channel channel = new Channel(agentAdress, ChannelCredentials.Insecure);
return new GameServerGrpcSdkService.GameServerGrpcSdkServiceClient(channel);
}
}
public static GseGrpcSdkService.GseGrpcSdkServiceClient GseClient
{
get
{
Channel channel = new Channel(agentAdress, ChannelCredentials.Insecure);
return new GseGrpcSdkService.GseGrpcSdkServiceClient(channel);
}
}
}
```

Demo for C#

1. Click here to download the code of the Demo for C#.
2. Generate the gRPC code.
   As the gRPC code has already been generated in the `proto/csharp-demo` directory of the Demo for C#, you do not need to generate it again.
3. Launch the server for GSE to call.

- Implement the server.
  `gameserversdk.cs` in the `csharp-demo/api` directory implements three server APIs.
- Run the server.
  `Program.cs` in the `csharp-demo` directory launches `GrpcServer`.

4. Connect the client to the gRPC server of GSE.
   - Implement the client.
     `GseManager.cs` in the `csharp-demo/Models` directory implements nine client APIs.
   - Connect to the server.
     Create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.
5. Compile and run the program.
   i. Generate the executable file and dependencies

```
dotnet publish -c Release -r linux-x64 --self-contained true
```

   The above operation will generate all the dependent files needed to generate and package the asset package in the `csharp-demo/bin/Release/netcoreapp3.1/linux-x64` directory, which contains the executable file `csharpdemo` used to run the service.
   - Copy the pre-request script `install.sh`

```
chmod u+x install.sh
cp install.sh bin/Release/netcoreapp3.1/linux-x64
```

   - Package the GSE asset package

```
cd csharp-demo/bin/Release/netcoreapp3.1/linux-x64
zip -r csharpdemo.zip *
```

     The packaged `csharpdemo.zip` is the asset package needed by GSE. Configure the launch path as `csharpdemo` with no launch parameter needed.
   - Create a server fleet and deploy the asset package on it. After that, you can perform various operations such as scaling.

gRPC Go Tutorial

Last updated : 2021-11-17 18:06:09

Installing gRPC

1. To use gRPC Go, you need to install the latest major release of Go first.
2. Install the protocol buffer compiler protoc3.
3. Install the Go plugin in the protocol buffer compiler.
   - Run the following command to install the protocol buffer compiler plugin for Go (protoc-gen-go):

     ```
     $ export GO111MODULE=on # Enable module mode
     $ go get github.com/golang/protobuf/protoc-gen-go
     ```
   - Update the path so that the protocol buffer compiler can find the Go plugin:

     ```
     $ export PATH="$PATH:$(go env GOPATH)/bin"
     ```

Note
For more information on the installation process, see Installing Go and Installing Protocol Buffer Compiler.

Defining Service

gRPC uses Protocol Buffers to define a service: an RPC service specifies methods that can be called remotely by using parameters and return types.

Note
We provide the .proto files for service definition. You can click here to directly download them with no need to generate them by yourself.

Generating gRPC Code

1. After defining the service, you can use protoc (protocol buffer compiler) to generate the client and server code (in any language supported by gRPC).

2. The generated code includes the client stub and the abstract APIs to be implemented by the server.

3. Steps for generating gRPC code:
   In the `proto` directory, run:

```
protoc --go_out=plugins=grpc:. *.proto
```

   to automatically generate the `go_package` path that contains proto. You can modify the `go_package` path as needed but not the package.

```
## Game Process Integration Process
![](https://main.qcloudimg.com/raw/96018551bc88c71a02333b1f197b3111.png)
#### Game server callback API list
| API Name | API Description |
|-----|----|
|[OnHealthCheck](https://intl.cloud.tencent.com/document/product/1055/37422)| Runs health chec
k |
|[OnStartGameServerSession](https://intl.cloud.tencent.com/document/product/1055/37423)| Recei
ves game server session |
|[OnProcessTerminate](https://intl.cloud.tencent.com/document/product/1055/37424)| Ends game p
rocess |
#### Game server active API list
| API Name | API Description |
|-----|----|
|[ProcessReady](https://intl.cloud.tencent.com/document/product/1055/37426)| Gets process read
y |
|[ActivateGameServerSession](https://intl.cloud.tencent.com/document/product/1055/37427)| Acti
vates game server session |
|[AcceptPlayerSession](https://intl.cloud.tencent.com/document/product/1055/37428)| Receives p
layer session |
|[RemovePlayerSession](https://intl.cloud.tencent.com/document/product/1055/37429)| Removes pl
ayer session |
|[DescribePlayerSessions](https://intl.cloud.tencent.com/document/product/1055/37430)| Gets pl
ayer session list |
|[UpdatePlayerSessionCreationPolicy](https://intl.cloud.tencent.com/document/product/1055/3743
1)| Updates player session creation policy |
|[TerminateGameServerSession](https://intl.cloud.tencent.com/document/product/1055/37432)| End
s game server session |
|[ProcessEnding](https://intl.cloud.tencent.com/document/product/1055/37434)| Ends process |
|[ReportCustomData](https://intl.cloud.tencent.com/document/product/1055/37435)| Reports custo
```

```
m data |
#### Others
When the game process uses gRPC to call a game server active API, you need to add two fields t
o `meta` of the gRPC request.
| Field | Description | Type |
| --------- | ------------------------------------- | ------ |
| pid | `pid` of the current game process | string |
| requestId | `requestId` of the current request, which is used to uniquely identify a request
| string |
1. Generally, after the server is initialized, the process will check itself to see whether it
can provide services, and the game server will call the `ProcessReady` API to notify GSE that
the process is ready to host a game server session. After receiving the notification, GSE will
change the status of the server instance to "Active".
```

Go

func (g *gsemanager) ProcessReady(logPath []string, clientPort int32, grpcPort int32) error {
logger.Info("start to processready", zap.Any("logPath", logPath), zap.Int32("clientPort", clientPort),

```
zap.Int32("grpcPort", grpcPort))
```

req := &grpcsdk.ProcessReadyRequest{

```
// Log path
LogPathsToUpload: logPath,
// Set the ports
ClientPort: clientPort,
GrpcPort: grpcPort,
```

}

_, err := g.rpcClient.ProcessReady(g.getContext(), req)
if err != nil {

```
logger.Info("ProcessReady fail", zap.Error(err))
return err
```

}

// Ready to provide services
logger.Info("ProcessReady success")

return nil
}

```
2. After the process is ready, GSE will call the `OnHealthCheck` API to perform a health check
on the game server every minute. If the health check fails three consecutive times, the proces
s will be considered to be unhealthy, and no game server sessions will be assigned to it.
```

Go

```
func _GameServerGrpcSdkService_OnHealthCheck_Handler(srv interface{}, ctx context.Context,
dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
in := new(HealthCheckRequest)
if err := dec(in); err != nil {
    return nil, err
}
if interceptor == nil {
    return srv.(GameServerGrpcSdkServiceServer).OnHealthCheck(ctx, in)
}
info := &grpc.UnaryServerInfo{
    Server: srv,
    FullMethod: "/tencentcloud.gse.grpcsdk.GameServerGrpcSdkService/OnHealthCheck",
}
handler := func(ctx context.Context, req interface{}) (interface{}, error) {
    return srv.(GameServerGrpcSdkServiceServer).OnHealthCheck(ctx, req.(*HealthCheckRequest))
}
return interceptor(ctx, in, info, handler)
}
```

```
3. Because the client calls the [CreateGameServerSession](https://intl.cloud.tencent.com/docum
ent/product/1055/37139) API to create a game server session and assigns it to a process, GSE w
```

```
ill be triggered to call the `onStartGameServerSession` API for the process and change the sta
tus of `GameServerSession` to "Activating".
```

Go

```
func _GameServerGrpcSdkService_OnStartGameServerSession_Handler(srv interface{}, ctx
context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor)
(interface{}, error) {
in := new(StartGameServerSessionRequest)
if err := dec(in); err != nil {
    return nil, err
}
if interceptor == nil {
    return srv.(GameServerGrpcSdkServiceServer).OnStartGameServerSession(ctx, in)
}
info := &grpc.UnaryServerInfo{
    Server: srv,
    FullMethod: "/tencentcloud.gse.grpcsdk.GameServerGrpcSdkService/OnStartGameServerSession",
}
handler := func(ctx context.Context, req interface{}) (interface{}, error) {
    return srv.(GameServerGrpcSdkServiceServer).OnStartGameServerSession(ctx, req.(*StartGameServe
    rSessionRequest))
}
return interceptor(ctx, in, info, handler)
}
```

```
4. After the game server receives `onStartGameServerSession`, you need to handle the logic or
resource allocation by yourself. After everything is ready, the game server will call the `Act
ivateGameServerSession` API to notify GSE that the game server session has been assigned to a
process and is ready to receive player requests and will change the server status to "Active".
```

Go

func (g *gsemanager) ActivateGameServerSession(gameServerSessionId string, maxPlayers int32) error {
logger.Info("start to ActivateGameServerSession", zap.String("gameServerSessionId", gameServerSessionId),

```
zap.Int32("maxPlayers", maxPlayers))
```

req := &grpcsdk.ActivateGameServerSessionRequest{

```
GameServerSessionId: gameServerSessionId,
MaxPlayers: maxPlayers,
```

}

_, err := g.rpcClient.ActivateGameServerSession(g.getContext(), req)
if err != nil {

```
logger.Error("ActivateGameServerSession fail", zap.Error(err))
return err
```

}

logger.Info("ActivateGameServerSession success")
return nil
}

```
5. After the client calls the [JoinGameServerSession](https://intl.cloud.tencent.com/document/product/1055/39130) API for the player to join, the game server will call the `AcceptPlayerSession` API to verify the validity of the player. If the connection is accepted, the status of `PlayerSession` will be set to "Active". If the client receives no response within 60 seconds after calling the `JoinGameServerSession` API, it will change the status of `PlayerSession` to "Timeout" and then call `JoinGameServerSession` again.
```

func (g *gsemanager) AcceptPlayerSession(playerSessionId string) (grpcsdk.GseResponse, error) {
logger.Info("start to AcceptPlayerSession", zap.String("playerSessionId", playerSessionId))
req := &grpcsdk.AcceptPlayerSessionRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
PlayerSessionId: playerSessionId,
```

}

return g.rpcClient.AcceptPlayerSession(g.getContext(), req)
}

```
6. After the game ends or the player exits, the game server will call the `RemovePlayerSession` API to remove the player, change the status of `playersession` to "Completed", and reserve the player slot in the game server session.
```

Go

func (g *gsemanager) RemovePlayerSession(playerSessionId string) (grpcsdk.GseResponse, error)
{
logger.Info("start to RemovePlayerSession", zap.String("playerSessionId", playerSessionId))
req := &grpcsdk.RemovePlayerSessionRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
PlayerSessionId: playerSessionId,
```

}

return g.rpcClient.RemovePlayerSession(g.getContext(), req)
}

```
7. After a game server session (a game battle or a service) ends, the game server will call the `TerminateGameServerSession` API to end the `GameServerSession` and change its status to `Terminated`.
```

Go

func (g *gsemanager) TerminateGameServerSession() (grpcsdk.GseResponse, error) {
logger.Info("start to TerminateGameServerSession")
req := &grpcsdk.TerminateGameServerSessionRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
```

}

return g.rpcClient.TerminateGameServerSession(g.getContext(), req)
}

```
8. In case of health check failure or reduction, GSE will call the `OnProcessTerminate` API to end the game process. The reduction will be triggered according to the [protection policy](https://intl.cloud.tencent.com/document/product/1055/36675) configured in the GSE console.
```

Go

func _GameServerGrpcSdkService_OnProcessTerminate_Handler(srv interface{}, ctx context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
in := new(ProcessTerminateRequest)
if err := dec(in); err != nil {

```
return nil, err
```

}
if interceptor == nil {

```
return srv.(GameServerGrpcSdkServiceServer).OnProcessTerminate(ctx, in)
```

}
info := &grpc.UnaryServerInfo{

```
Server: srv,
FullMethod: "/tencentcloud.gse.grpcsdk.GameServerGrpcSdkService/OnProcessTerminate",
```

}
handler := func(ctx context.Context, req interface{}) (interface{}, error) {

```
return srv.(GameServerGrpcSdkServiceServer).OnProcessTerminate(ctx, req.(*ProcessTerminateRequest))
```

}
return interceptor(ctx, in, info, handler)

}

```
9. The game server calls the `ProcessEnding` API to end the process immediately, change the server process status to "Terminated", and repossess the resources.
```

Go

// Active call: a game battle corresponds to a process. The `ProcessEnding` API will be actively called after the game battle ends
// Passive call: in case of reduction, process exception, or health check failure, the `ProcessEnding` API will be called passively according to the protection policy. If a full protection or time-period protection policy is configured, it is required to determine whether there are any players in the game server session before the passive call can be made
func (g *gsemanager) ProcessEnding() (*grpcsdk.GseResponse, error) {
logger.Info("start to ProcessEnding")
req := &grpcsdk.ProcessEndingRequest{
}

return g.rpcClient.ProcessEnding(g.getContext(), req)
}

```
10. The game server calls the `DescribePlayerSessions` API to get the information of the player in the game server session (which is optional based on your actual business needs).
```

Go

func (g *gsemanager) DescribePlayerSessions(gameServerSessionId, playerId, playerSessionId, playerSessionStatusFilter, nextToken string,limit int32) (*grpcsdk.DescribePlayerSessionsResponse, error) {
logger.Info("start to DescribePlayerSessions", zap.String("gameServerSessionId", gameServerSessionId),

```
zap.String("playerId", playerId), zap.String("playerSessionId", playerSessionId),
zap.String("playerSessionStatusFilter", playerSessionStatusFilter), zap.String("nextToken", nextToken),
zap.Int32("limit", limit))
```

req := &grpcsdk.DescribePlayerSessionsRequest{

```
GameServerSessionId: gameServerSessionId,
PlayerId: playerId,
PlayerSessionId: playerSessionId,
PlayerSessionStatusFilter: playerSessionStatusFilter,
NextToken: nextToken,
Limit: limit,
```

}

return g.rpcClient.DescribePlayerSessions(g.getContext(), req)
}

```
11. The game server calls the `UpdatePlayerSessionCreationPolicy` API to update the player session creation policy and set whether to accept new players, i.e., whether to allow new players to join a game session (which is optional based on your actual business needs).
```

Go

func (g *gsemanager) UpdatePlayerSessionCreationPolicy(newPolicy string) (*grpcsdk.GseResponse, error) {
logger.Info("start to UpdatePlayerSessionCreationPolicy", zap.String("newPolicy", newPolicy))
req := &grpcsdk.UpdatePlayerSessionCreationPolicyRequest{

```
GameServerSessionId: g.gameServerSession.GameServerSessionId,
NewPlayerSessionCreationPolicy: newPolicy,
```

}

return g.rpcClient.UpdatePlayerSessionCreationPolicy(g.getContext(), req)
}

```
12. The game server calls the `ReportCustomData` API to notify GSE of the custom data (which is optional based on your actual business needs).
```

Go

func (g *gsemanager) ReportCustomData(currentCustomCount, maxCustomCount int32) (*grpcsdk.GseResponse, error) {
logger.Info("start to UpdatePlayerSessionCreationPolicy", zap.Int32("currentCustomCount", currentCustomCount),

```
zap.Int32("maxCustomCount", maxCustomCount))
```

req := &grpcsdk.ReportCustomDataRequest{

```
CurrentCustomCount: currentCustomCount,
MaxCustomCount: maxCustomCount,
```

}

return g.rpcClient.ReportCustomData(g.getContext(), req)
}

```
## Launching Server for GSE to Call
Server running: launch `GrpcServer`.
```

Go

func (s *rpcService) StartGrpcServer() {
listen, err := net.Listen("tcp", "127.0.0.1:")
if err != nil {

```
logger.Fatal("grpc fail to listen", zap.Error(err))
```

}

addr := listen.Addr().String()
portStr := strings.Split(addr, ":")[1]
s.grpcPort, err = strconv.Atoi(portStr)
if err != nil {

```
logger.Fatal("grpc fail to get port",zap.Error(err))
```

}

logger.Info("grpc listen port is", zap.Int("port", s.grpcPort))

grpcServer := grpc.NewServer()
grpcsdk.RegisterGameServerGrpcSdkServiceServer(grpcServer, s)
logger.Info("start grpc server success")
go grpcServer.Serve(listen)
}

```
## Connecting Client to gRPC Server of GSE
Server connecting: create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.
```

Go

const (

```
localhost = "127.0.0.1"
agentPort = 5758
```

)
type gsemanager struct {

```
pid string
gameServerSession *grpcsdk.GameServerSession
terminationTime int64
rpcClient grpcsdk.GseGrpcSdkServiceClient
```

}

```
## Demo for Go
1. [Click here](https://gsegrpcdemo-1301007756.cos.ap-guangzhou.myqcloud.com/go-demo.zip) to download the code of the Demo for Go.
2. Generate the gRPC code.
As the gRPC code has already been generated in the `go-demo/grpcsdk` directory of the Demo for Go, you do not need to generate it again.
3. Launch the server for GSE to call.
- Implement the server.
`grpcserver.go` in the `go-demo/api` directory implements three server APIs.
- Run the server.
`grpcserver.go` in the `go-demo/api` directory launches `GrpcServer`.
4. Connect the client to the gRPC server of GSE.
- Implement the client.
`gsemanager.go` in the `go-demo/gsemanager` directory implements nine client APIs.
- Connect to the server.
```

```
Create a gRPC channel, specify the host name and server port to connect to, and use this chann
el to create a stub instance.
5. Compile and run the project.
1. In the `go-demo` directory, run
```

go mod vendor

```
to generate the vendor directory.


◦ Run the compile command:
```

go build -mod=vendor main.go

```
to generate the corresponding go-demo executable file main.go.
```

4. Package the executable file `main.go` as an [asset package](https://intl.cloud.tencent.com/document/product/1055/36674) and configure the launch path as `main` with no launch parameter needed.
5. [Create a server fleet](https://intl.cloud.tencent.com/document/product/1055/36675) and deploy the asset package on it. After that, you can perform various operations such as [scaling](https://intl.cloud.tencent.com/document/product/1055/37445).

gRPC Java Tutorial

Last updated : 2021-11-17 18:06:09

Installing gRPC

1. gRPC Java does not require other tools except JDK.
2. Install the gRPC Java SNAPSHOT library locally, including the code generation plugin.

> Note :
> For more information on the installation process, please see Installing gRPC Java.

Defining Service

gRPC uses Protocol Buffers to define a service: an RPC service specifies methods that can be called remotely by using parameters and return types.

> Note
> We provide the .proto files for service definition. You can click here to directly download them with no need to generate them by yourself.

Generating gRPC Code

1. After defining the service, you can use protoc (protocol buffer compiler) to generate the client and server code (in any language supported by gRPC).
2. The generated code includes the client stub and the abstract APIs to be implemented by the server.
3. Methods for generating gRPC code:
   - Method 1. Execute the script under `java-demo/src/main/proto` . You need to download `protoc` and `protoc-gen-grpc-java` generation tools from the gRPC website:

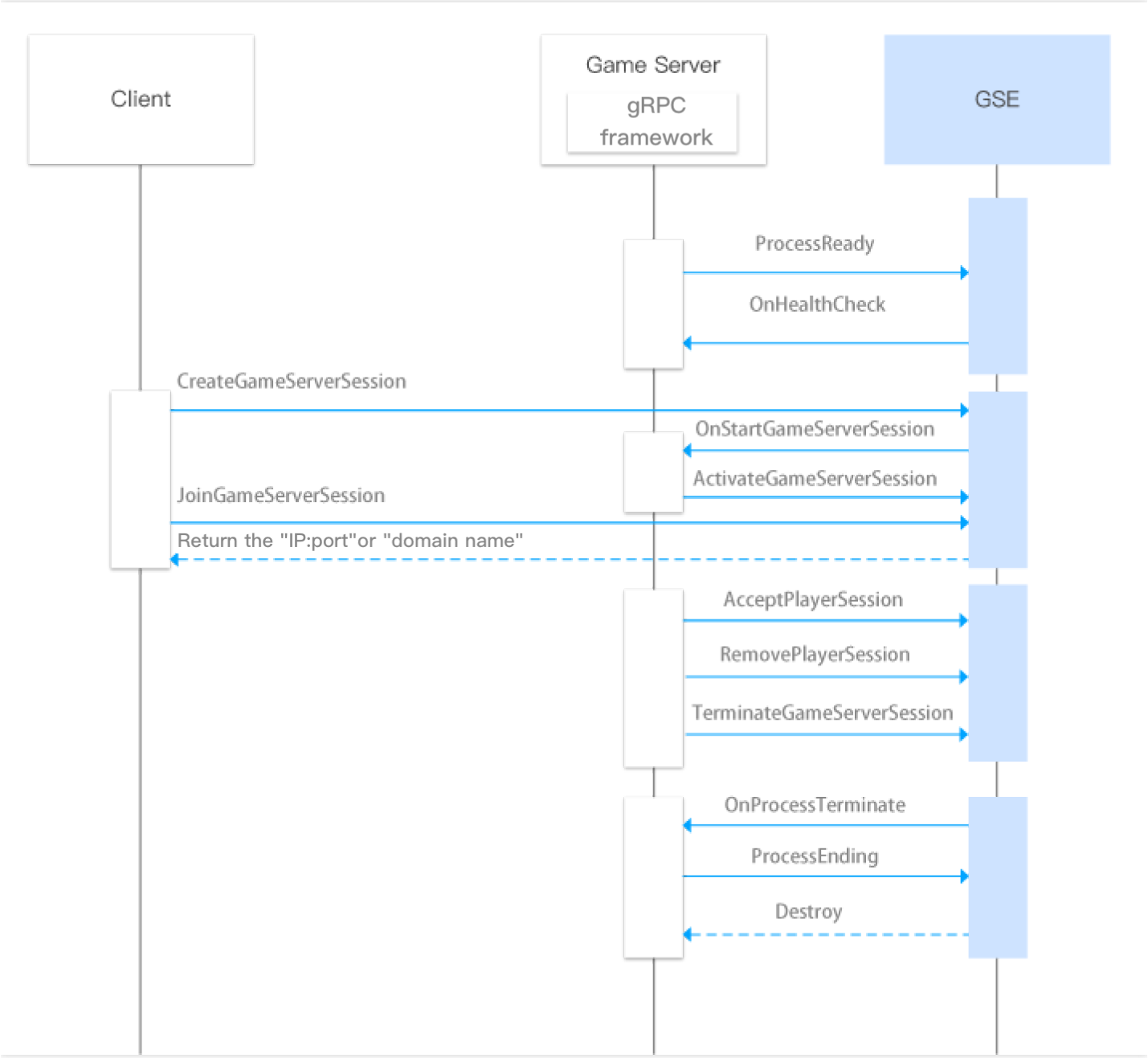
```
sh gen_pb.sh
```

```
protoc --java_out=../java --proto_path=. GameServerGrpcSdkService.proto
protoc --plugin=protoc-gen-grpc-java=`which protoc-gen-grpc-java` --grpc-java_out=../java --
proto_path=. GameServerGrpcSdkService.proto
protoc --java_out=../java --proto_path=. GseGrpcSdkService.proto
protoc --plugin=protoc-gen-grpc-java=`which protoc-gen-grpc-java` --grpc-java_out=../java --
proto_path=. GseGrpcSdkService.proto
```

- Method 2. Use the Maven tool to generate gRPC code by adding a Maven plugin for compiling gRPC code to Maven. For more information, please see here.

```
<build>
<extensions>
<extension>
<groupid>kr.motd.maven</groupid>
<artifactid>os-maven-plugin</artifactid>
<version>1.6.2</version>
</extension>
</extensions>
<plugins>
<plugin>
<groupid>org.xolstice.maven.plugins</groupid>
<artifactid>protobuf-maven-plugin</artifactid>
<version>0.6.1</version>
<configuration>
<protocartifact>com.google.protobuf:protoc:3.12.0:exe:${os.detected.classifier}</protocartif
act>
<pluginid>grpc-java</pluginid>
<pluginartifact>io.grpc:protoc-gen-grpc-java:1.30.2:exe:${os.detected.classifier}</pluginart
ifact>
</configuration>
<executions>
<execution>
<goals>
<goal>compile</goal>
<goal>compile-custom</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

Game Process Integration Process



Game server callback API list

API Name	API Description
OnHealthCheck	Runs health check
OnStartGameServerSession	Receives game server session
OnProcessTerminate	Ends game process

Game server active API list

API Name	API Description
ProcessReady	Gets process ready
ActivateGameServerSession	Activates game server session
AcceptPlayerSession	Receives player session
RemovePlayerSession	Removes player session
DescribePlayerSessions	Gets player session list
UpdatePlayerSessionCreationPolicy	Updates player session creation policy
TerminateGameServerSession	Ends game server session
ProcessEnding	Ends process
ReportCustomData	Reports custom data

Others

When the game process uses gRPC to call a game server active API, you need to add two fields to `meta` of the gRPC request.

Field	Description	Type
pid	`pid` of the current game process	string
requestId	`requestId` of the current request, which is used to uniquely identify a request	string

1. Generally, after the server is initialized, the process will check itself to see whether it can provide services, and the game server will call the `ProcessReady` API to notify GSE that the process is ready to host a game server session. After receiving the notification, GSE will change the status of the server instance to "Active".

```
public GseResponseBo processReady(ProcessReadyRequestBo request) {
logger.info("processReady request=" + new Gson().toJson(request));
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ProcessReadyRequest rpcRequest = GseGrpcSdkServiceOuterClass.Proce
ssReadyRequest
// Set the ports.
.newBuilder().setClientPort(request.getClientPort())
.setGrpcPort(request.getGrpcPort())
// Log path.
```

```
.addAllLogPathsToUpload(request.getLogPathsToUploadList()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().processReady(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
// Ready to provide services.
logger.info("processReady response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

2. After the process is ready, GSE will call the `OnHealthCheck` API to perform a health check on the game server every minute. If the health check fails three consecutive times, the process will be considered to be unhealthy, and no game server sessions will be assigned to it.

```
public boolean onHealthCheck() {
// Add game server logic for health check.
boolean res = getGrpcServiceConfig().getGseGrpcSdkServiceClient().isProcessHealth();
logger.info("onHealthCheck status=" + res);
return res;
}
```

3. Because the client calls the CreateGameServerSession API to create a game server session and assigns it to a process, GSE will be triggered to call the `onStartGameServerSession` API for the process and change the status of `GameServerSession` to "Activating".

```
public GseResponseBo onStartGameServerSession(GameServerSessionBo gameServerSessionBo) {
logger.info("onStartGameServerSession gameServerSession=" + new Gson().toJson(gameServerSessio
nBo));
// Add the game server logic used to launch the game server session.
// Save the game server sessions
getGrpcServiceConfig().getGseGrpcSdkServiceClient().onStartGameServerSession(gameServerSession
Bo);
// Activate the game server sessions
ActivateGameServerSessionRequestBo activateRequest = new ActivateGameServerSessionRequestBo();
activateRequest.setGameServerSessionId(gameServerSessionBo.getGameServerSessionId());
activateRequest.setMaxPlayers(gameServerSessionBo.getMaxPlayers());
getGrpcServiceConfig().getGseGrpcSdkServiceClient().activateGameServerSession(activateReques
t);
// Add the final logic here.
return createResponseBo(0, "SUCCESS");
}
```

4. After the game server receives `onStartGameServerSession`, you need to handle the logic or resource allocation by yourself. After everything is ready, the game server will call the `ActivateGameServerSession` API to notify GSE that the game server session has been assigned to a process and is ready to receive player requests and will change the server status to "Active".

```
public GseResponseBo activateGameServerSession(ActivateGameServerSessionRequestBo request) {
logger.info("activateGameServerSession request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ActivateGameServerSessionRequest rpcRequest = GseGrpcSdkServiceOut
erClass.ActivateGameServerSessionRequest
.newBuilder().setMaxPlayers(request.getMaxPlayers())
.setGameServerSessionId(gameServerSessionBo.getGameServerSessionId()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().activateGameServerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("activateGameServerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

5. After the client calls the JoinGameServerSession API for the player to join, the game server will call the `AcceptPlayerSession` API to verify the validity of the player. If the connection is accepted, the status of `PlayerSession` will be set to "Active". If the client receives no response within 60 seconds after calling the `JoinGameServerSession` API, it will change the status of `PlayerSession` to "Timeout" and then call `JoinGameServerSession` again.

```
public GseResponseBo acceptPlayerSession(PlayerSessionRequestBo request) {
logger.info("acceptPlayerSession request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.AcceptPlayerSessionRequest rpcRequest = GseGrpcSdkServiceOuterClas
s.AcceptPlayerSessionRequest
```

```
.newBuilder().setGameServerSessionId(gameServerSessionBo.getGameServerSessionId())
.setPlayerSessionId(request.getPlayerSessionId()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().acceptPlayerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("acceptPlayerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

6. After the game ends or the player exits, the game server will call the `RemovePlayerSession` API to remove the player, change the status of `playersession` to "Completed", and reserve the player slot in the game server session.

```
public GseResponseBo removePlayerSession(PlayerSessionRequestBo request) {
logger.info("removePlayerSession request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.RemovePlayerSessionRequest rpcRequest = GseGrpcSdkServiceOuterClas
s.RemovePlayerSessionRequest
.newBuilder().setGameServerSessionId(gameServerSessionBo.getGameServerSessionId())
.setPlayerSessionId(request.getPlayerSessionId()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().removePlayerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("removePlayerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

7. After a game server session (a game battle or a service) ends, the game server will call the `TerminateGameServerSession` API to end the `GameServerSession` and change its status to `Terminated`.

```
public GseResponseBo terminateGameServerSession(String gameServerSessionId) {
logger.info("terminateGameServerSession request=" + gameServerSessionId);
if (StringUtils.isEmpty(gameServerSessionId) && gameServerSessionBo != null
&& !StringUtils.isEmpty(gameServerSessionBo.getGameServerSessionId())) {
gameServerSessionId = gameServerSessionBo.getGameServerSessionId();
}
if (StringUtils.isEmpty(gameServerSessionId)) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.TerminateGameServerSessionRequest rpcRequest = GseGrpcSdkServiceOu
terClass.TerminateGameServerSessionRequest
.newBuilder().setGameServerSessionId(gameServerSessionId).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().terminateGameServerSession(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("terminateGameServerSession response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

8. In case of health check failure or reduction, GSE will call the `OnProcessTerminate` API to end the game process. The reduction will be triggered according to the protection policy configured in the GSE console.

```
public GseResponseBo onProcessTerminate(long terminationTime) {
logger.info("onProcessTerminate terminationTime=" + terminationTime);
// It is possible to end the game server now.
return createResponseBo(0, "SUCCESS");
}
```

9. The game server calls the `ProcessEnding` API to end the process immediately, change the server process status to "Terminated", and repossess the resources.

```
// Active call: a game battle corresponds to a process. The `ProcessEnding` API will be active
ly called after the game battle ends
// Passive call: in case of reduction, process exception, or health check failure, the `Proces
sEnding` API will be called passively according to the protection policy. If a full protection
or time-period protection policy is configured, it is required to determine whether there are
```

```
any players in the game server session before the passive call can be made
public GseResponseBo processEnding() {
logger.info("processEnding begin");
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ProcessEndingRequest rpcRequest = GseGrpcSdkServiceOuterClass.Proc
essEndingRequest
.newBuilder().build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().processEnding(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("processEnding response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

0. The game server calls the `DescribePlayerSessions` API to get the information of the player in the game server session (which is optional based on your actual business needs).

```
public DescribePlayerSessionsResponseBo describePlayerSessions(DescribePlayerSessionsRequestBo
request) {
logger.info("describePlayerSessions request=" + new Gson().toJson(request));
if (StringUtils.isEmpty(request.getGameServerSessionId()) &&
gameServerSessionBo != null && !StringUtils.isEmpty(gameServerSessionBo.getGameServerSessionId
())) {
request.setGameServerSessionId(gameServerSessionBo.getGameServerSessionId());
}
GseGrpcSdkServiceOuterClass.DescribePlayerSessionsRequest rpcRequest = GseGrpcSdkServiceOuterC
lass.DescribePlayerSessionsRequest
.newBuilder().setGameServerSessionId(request.getGameServerSessionId())
.setLimit(request.getLimit())
.setNextToken(request.getNextToken())
.setPlayerId(request.getPlayerId())
.setPlayerSessionId(request.getPlayerSessionId())
.setPlayerSessionStatusFilter(request.getPlayerSessionStatusFilter()).build();
GseGrpcSdkServiceOuterClass.DescribePlayerSessionsResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().describePlayerSessions(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return null;
}
logger.info("describePlayerSessions response=" + rpcResponse.toString());
```

```
return toPlayerSessionsResponseBo(rpcResponse);
}
```

1. The game server calls the `UpdatePlayerSessionCreationPolicy` API to update the player session creation policy and set whether to accept new players, i.e., whether to allow new players to join a game session (which is optional based on your actual business needs).

```
public GseResponseBo updatePlayerSessionCreationPolicy(UpdatePlayerSessionCreationPolicyReques
tBo request) {
logger.info("updatePlayerSessionCreationPolicy request=" + new Gson().toJson(request));
if (gameServerSessionBo == null) {
return createResponseBo(Constants.gameServerSessionExpectCode, "no game server session found."
);
}
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.UpdatePlayerSessionCreationPolicyRequest rpcRequest = GseGrpcSdkSe
rviceOuterClass.UpdatePlayerSessionCreationPolicyRequest
.newBuilder().setGameServerSessionId(gameServerSessionBo.getGameServerSessionId())
.setNewPlayerSessionCreationPolicy(request.getNewPlayerSessionCreationPolicy()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().updatePlayerSessionCreationPolicy(rpcRequest);
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("updatePlayerSessionCreationPolicy response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

2. The game server calls the `ReportCustomData` API to notify GSE of the custom data (which is optional based on your actual business needs).

```
public GseResponseBo reportCustomData(ReportCustomDataRequestBo request) {
logger.info("reportCustomData request=" + new Gson().toJson(request));
GseResponseBo responseBo = new GseResponseBo();
GseGrpcSdkServiceOuterClass.ReportCustomDataRequest rpcRequest = GseGrpcSdkServiceOuterClass.R
eportCustomDataRequest
.newBuilder()
.setCurrentCustomCount(request.getCurrentCustomCount())
.setMaxCustomCount(request.getMaxCustomCount()).build();
GseGrpcSdkServiceOuterClass.GseResponse rpcResponse;
try {
rpcResponse = getGseGrpcSdkServiceClient().reportCustomData(rpcRequest);
```

```
} catch (StatusRuntimeException e) {
logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
return createRpcFailedResponseBo(e.getStatus());
}
logger.info("reportCustomData response=" + rpcResponse.toString());
return createResponseBoByRpcResponse(rpcResponse);
}
```

Launching Server for GSE to Call

Server running: launch `GrpcServer` .

```
@Bean(name = "grpcService", initMethod = "startup", destroyMethod = "shutdown")
public GrpcService getGrpcService() {
GrpcServiceConfig grpcServiceConfig = new GrpcServiceConfig();
grpcServiceConfig.setGseGrpcSdkServiceClient(gseGrpcSdkServiceClient);
grpcServiceConfig.setGameServerGrpcPort(gameServerGrpcPort);
grpcServiceConfig.setGameServerToClientPort(gameServerToClientPort);
grpcServiceConfig.setTargetAddress(targetAddress);
grpcServiceConfig.setGameServerUploadLogPath(gameServerUploadLogPath);
GrpcService grpcService = new GrpcService(grpcServiceConfig);
return grpcService;
}
```

Connecting Client to gRPC Server of GSE

Server connecting: create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.

```
public GseGrpcSdkServiceGrpc.GseGrpcSdkServiceBlockingStub getGseGrpcSdkServiceClient() {
// The "channel" here is a channel instead of a `ManagedChannel`; therefore, it is not the respon
sibility of the code to shut it down.
// Pass the channel to the code to make it easier for the code to test and reuse the channel.
if (blockingStub == null) {
managedChannel = getGrpcChannel(targetAddress);
blockingStub = GseGrpcSdkServiceGrpc.newBlockingStub(managedChannel);
}
if (managedChannel.isShutdown() || managedChannel.isTerminated()) {
managedChannel.shutdownNow();
managedChannel = getGrpcChannel(targetAddress);
blockingStub = GseGrpcSdkServiceGrpc.newBlockingStub(managedChannel);
```

```
}
return blockingStub;
}
```

Demo for Java

1. Click here to download the code of the Demo for Java.
2. Generate the gRPC code.
   As the gRPC code has already been generated in the `java-demo/src/main/java/tencentcloud` directory of the Demo for Java, you do not need to generate it again.
3. Launch the server for GSE to call.
   - Implement the server.
     `GameServerGrpcCallbackImpl.java` in the `java-demo/src/main/java/com/tencentcloud/gse/gameserver/service/gamelogic/impl` directory implements three server APIs.
   - Run the server.
     `GameServerConfig.java` in the `java-demo/src/main/java/com/tencentcloud/gse/gameserver/config` directory launches `GrpcServer` .
4. Connect the client to the gRPC server of GSE.
   - Implement the client.
     `GseGrpcSdkServiceClientImpl.java` in the `java-demo/src/main/java/com/tencentcloud/gse/gameserver/service/gsegrpc/impl` directory implements nine client APIs.
   - Connect to the server.
     Create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.
5. Compile and run the project.
   i. Java v1.8 or above is required. You can use `yum` to install `openjdk` on Linux:

   ```
   yum install -y java-1.8.0-openjdk
   ```

   - Download the code, use Maven to build and generate `gse-gameserver-demo.jar` in the `java-demo` directory, and run the following command to launch it:

   ```
   java -jar gse-gameserver-demo.jar
   ```

   - Package the executable file `gse-gameserver-demo.jar` as an asset package and configure the launch path as `java` and the launch parameter as `jar gse-gameserver-demo.jar` .

- Create a server fleet and deploy the asset package on it. After that, you can perform various operations such as scaling.

gRPC Lua Tutorial

Last updated : 2021-11-17 18:06:10

Installing gRPC

1. Install gRPC. The installation will generate an executable program `grpc_cpp_plugin`, which will be needed for generating gRPC code.
2. Install protocol buffers.

> Note
> For more information on the installation process, see Installing gRPC Lua and Installing Protocol Buffers.

Defining Service

gRPC uses Protocol Buffers to define a service: an RPC service specifies methods that can be called remotely by using parameters and return types.

> Note
> We provide the .proto files for service definition. You can click here to directly download them with no need to generate them by yourself.

Generating gRPC Code

1. After defining the service, you can use protoc (protocol buffer compiler) to generate the client and server code (in any language supported by gRPC).
2. The generated code includes the client stub and the abstract APIs to be implemented by the server.
3. Steps for generating gRPC code:
   The Demo for Lua relies on the C++ framework. Just like with the Demo for C++, in the `proto` directory, run:

```
protoc --cpp_out=. *.proto
```

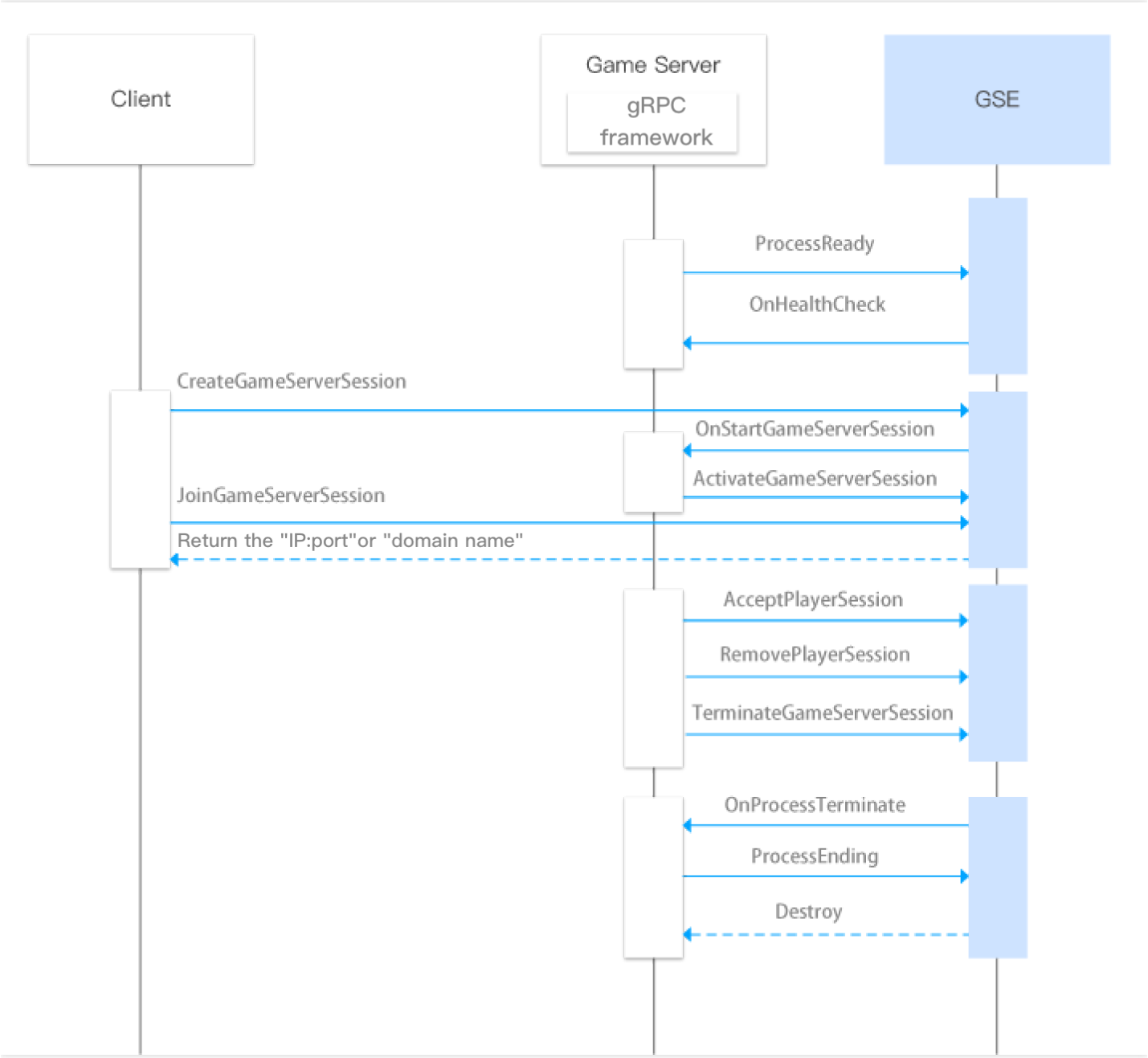
to generate the `pb.cc` and `pb.h` files.

```
protoc --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` *.proto
```

to generate the corresponding gRPC code.

Move the eight generated files to an appropriate location in the project.

Game Process Integration Process



Game server callback API list

API Name	API Description
OnHealthCheck	runs health check
OnStartGameServerSession	Receives game server session
OnProcessTerminate	Ends game process

Game server callback API list

API Name	API Description
ProcessReady	Gets process ready
ActivateGameServerSession	Activates game server session
AcceptPlayerSession	Receives player session
RemovePlayerSession	Removes player session
DescribePlayerSessions	Gets player session list
UpdatePlayerSessionCreationPolicy	Updates player session creation policy
TerminateGameServerSession	Ends game server session
ProcessEnding	Ends process
ReportCustomData	Reports custom data

Others

When the game process uses gRPC to call a game server active API, you need to add two fields to `meta` of the gRPC request.

Field	Description	Type
pid	`pid` of the current game process	string
requestId	`requestId` of the current request, which is used to uniquely identify a request	string

1. Generally, after the server is initialized, the process will check itself to see whether it can provide services, and the game server will call the `ProcessReady` API to notify GSE that the process is ready to host a game server session. After receiving the notification, GSE will change the status of the server instance to "Active".

```
static bool luaProcessReady(std::vector <std::string> &logPath, int clientPort, int grpcPort)
{
GseResponse reply;
// Log path. Set the ports.
Status status = GGseManager->ProcessReady(logPath, clientPort, grpcPort, reply);
// Ready to provide services
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
```

```
}
return true;
}
```

2. After the process is ready, GSE will call the `OnHealthCheck` API to perform a health check on the game server every minute. If the health check fails three consecutive times, the process will be considered to be unhealthy, and no game server sessions will be assigned to it.

```
Status GameServerGrpcSdkServiceImpl::OnHealthCheck(ServerContext* context, const HealthCheckRe
quest* request, HealthCheckResponse* reply)
{
healthStatus = GSESDK()->exec("return OnHealthCheck()");
std::cout << "healthStatus=" << healthStatus << std::endl;
reply->set_healthstatus(healthStatus);
return Status::OK;
}
```

3. Because the client calls the CreateGameServerSession API to create a game server session and assigns it to a process, GSE will be triggered to call the `onStartGameServerSession` API for the process and change the status of `GameServerSession` to "Activating".

```
Status GameServerGrpcSdkServiceImpl::OnStartGameServerSession(ServerContext* context, const St
artGameServerSessionRequest* request, GseResponse* reply)
{
auto gameServerSession = request->gameserversession();
GGseManager->SetGameServerSession(gameServerSession);
std::ostringstream o;
o << "return OnStartGameServerSession('" << gameServerSession.gameserversessionid() << "'," <<
gameServerSession.maxplayers() << ")";
std::string luaCmd = o.str();
bool res = GSESDK()->exec(luaCmd);
return Status::OK;
}
```

4. After the game server receives `onStartGameServerSession` , you need to handle the logic or resource allocation by yourself. After everything is ready, the game server will call the `ActivateGameServerSession` API to notify GSE that the game server session has been assigned to a process and is ready to receive player requests and will change the server status to "Active".

```
static bool luaActivateGameServerSession(const std::string &gameServerSessionId, int maxPlayer
s) {
```

```
GseResponse reply;
Status status = GGseManager->ActivateGameServerSession(gameServerSessionId, maxPlayers, repl
y);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

5. After the client calls the JoinGameServerSession API for the player to join, the game server will call the `AcceptPlayerSession` API to verify the validity of the player. If the connection is accepted, the status of `PlayerSession` will be set to "Active". If the client receives no response within 60 seconds after calling the `JoinGameServerSession` API, it will change the status of `PlayerSession` to "Timeout" and then call `JoinGameServerSession` again.

```
static bool luaAcceptPlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->AcceptPlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

6. After the game ends or the player exits, the game server will call the `RemovePlayerSession` API to remove the player, change the status of `playersession` to "Completed", and reserve the player slot in the game server session.

```
static bool luaRemovePlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->RemovePlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

7. After a game server session (a game battle or a service) ends, the game server will call the `TerminateGameServerSession` API to end the `GameServerSession` and change its status to `Terminated`.

```
static bool luaTerminateGameServerSession(const std::string &gameServerSessionId) {
GseResponse reply;
Status status = GGseManager->TerminateGameServerSession(gameServerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

8. In case of health check failure or reduction, GSE will call the `OnProcessTerminate` API to end the game process. The reduction will be triggered according to the protection policy configured in the GSE console.

```
Status GameServerGrpcSdkServiceImpl::OnProcessTerminate(ServerContext* context, const ProcessT
erminateRequest* request, GseResponse* reply)
{
auto terminationTime = request->terminationtime();
std::to_string(terminationTime));
std::ostringstream o;
o << "OnProcessTerminate(" << terminationTime << ")";
std::string luaCmd = o.str();
GSESDK()->execWithNilResult(luaCmd);
return Status::OK;
}
```

9. The game server calls the `ProcessEnding` API to end the process immediately, change the server process status to "Terminated", and repossess the resources.

```
// Active call: a game battle corresponds to a process. The `ProcessEnding` API will be active
ly called after the game battle ends
// Passive call: in case of reduction, process exception, or health check failure, the `Proces
sEnding` API will be called passively according to the protection policy. If a full protection
or time-period protection policy is configured, it is required to determine whether there are
any players in the game server session before the passive call can be made
static bool luaProcessEnding() {
GseResponse reply;
Status status = GGseManager->ProcessEnding(reply);
GSESDK()->setReplyStatus(status);
```

```
if (!status.ok()) {
return false;
}
return true;
}
```

0. The game server calls the `DescribePlayerSessions` API to get the information of the player in the game server session (which is optional based on your actual business needs).

```
static bool luaDescribePlayerSessions(const std::string &gameServerSessionId,
const std::string &playerId,
const std::string &playerSessionId,
const std::string &playerSessionStatusFilter, const std::string &nextToken,
int limit) {
DescribePlayerSessionsResponse reply;
Status status = GGseManager->DescribePlayerSessions(gameServerSessionId,playerId, playerSessio
nId, playerSessionStatusFilter, nextToken, limit, reply);
GSESDK()->setDescribePlayerSessionsResponse(reply);
if (!status.ok()) {
return false;
}
return true;
}
```

1. The game server calls the `UpdatePlayerSessionCreationPolicy` API to update the player session creation policy and set whether to accept new players, i.e., whether to allow new players to join a game session (which is optional based on your actual business needs).

```
static bool luaUpdatePlayerSessionCreationPolicy(const std::string &newpolicy) {
GseResponse reply;
Status status = GGseManager->UpdatePlayerSessionCreationPolicy(newpolicy, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

2. The game server calls the `ReportCustomData` API to notify GSE of the custom data (which is optional based on your actual business needs).

```
static bool luaReportCustomData(int currentCustomCount, int maxCustomCount) {
GseResponse reply;
Status status = GGseManager->ReportCustomData(currentCustomCount, maxCustomCount, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

Launching Server for GSE to Call

Server running: launch `GrpcServer` .

```
// Launch the gRPC server
std::thread tGrpc(&GameServerGrpcSdkServiceImpl::StartGrpcServer, GGameServerGrpcSdkService);
sem_wait(&(GGameServerGrpcSdkService->sem));
auto grpcPort = GGameServerGrpcSdkService->GetGrpcPort();
```

Connecting Client to gRPC Server of GSE

Server connecting: create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.

```
void GseManager::InitStub() {
auto channel = grpc::CreateChannel("127.0.0.1:5758", grpc::InsecureChannelCredentials());
stub_ = GseGrpcSdkService::NewStub(channel);
}
```

Demo for Lua

1. Click here to download the code of the Demo for Lua.
2. Generate the gRPC code.
   The Demo for Lua relies on the C++ framework, with gRPC code generated in the `cpp-demo/source/grpcsdk` directory, so there is no need to generate it again.
3. Launch the server for GSE to call.

- Implement the server.
  `grpcserver.cpp` in the `lua-demo/source/api` directory implements three server APIs.
- Run the server.
  `main.cpp` in the `lua-demo` directory launches `GrpcServer` .

4. Connect the client to the gRPC server of GSE.
   - Implement the client.
     `GSESdkHandleWrapper.cpp` in the `lua-demo/source/lua` directory implements nine client APIs.
   - Connect to the server.
     Create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.
5. Compile and run the project.
   i. Install CMake.
   - Install GCC v4.9 or above.
   - Install the LuaJIT and Boost development kits:

```
yum install -y luajit-devel
yum install -y boost-devel
yum install -y cmake
```

   - Download the code and run the following command in the `lua-demo` directory:

```
mkdir build
cd build
cmake ..
make
cp ../source/lua/gse.lua .
```

     The corresponding `lua-demo` executable file will be generated. Run `./lua-demo` to launch it.
   - Package the executable file `lua-demo.cpp` as an asset package and configure the launch path as `lua-demo` with no launch parameter needed.
   - Create a server fleet and deploy the asset package on it. After that, you can perform various operations such as scaling.

gRPC Node.js Tutorial

Last updated : 2022-01-21 11:48:44

Installing gRPC

1. Install gRPC. The installation will generate an executable program `grpc_cpp_plugin`, which will be needed for generating gRPC code.
2. Install protocol buffers.

> Note
> For more information on the installation process, see Installing gRPC Lua and Installing Protocol Buffers.

Defining Service

gRPC uses Protocol Buffers to define a service: an RPC service specifies methods that can be called remotely by using parameters and return types.

> Note
> We provide the .proto files for service definition. You can click here to directly download them with no need to generate them by yourself.

Generating gRPC Code

1. After defining the service, you can use protoc (protocol buffer compiler) to generate the client and server code (in any language supported by gRPC).

2. The generated code includes the client stub and the abstract APIs to be implemented by the server.

3. Steps for generating gRPC code:

i. The Demo for Lua relies on the C++ framework. Just like with the Demo for C++, in the `proto` directory, run:

```
protoc --cpp_out=. *.proto
```

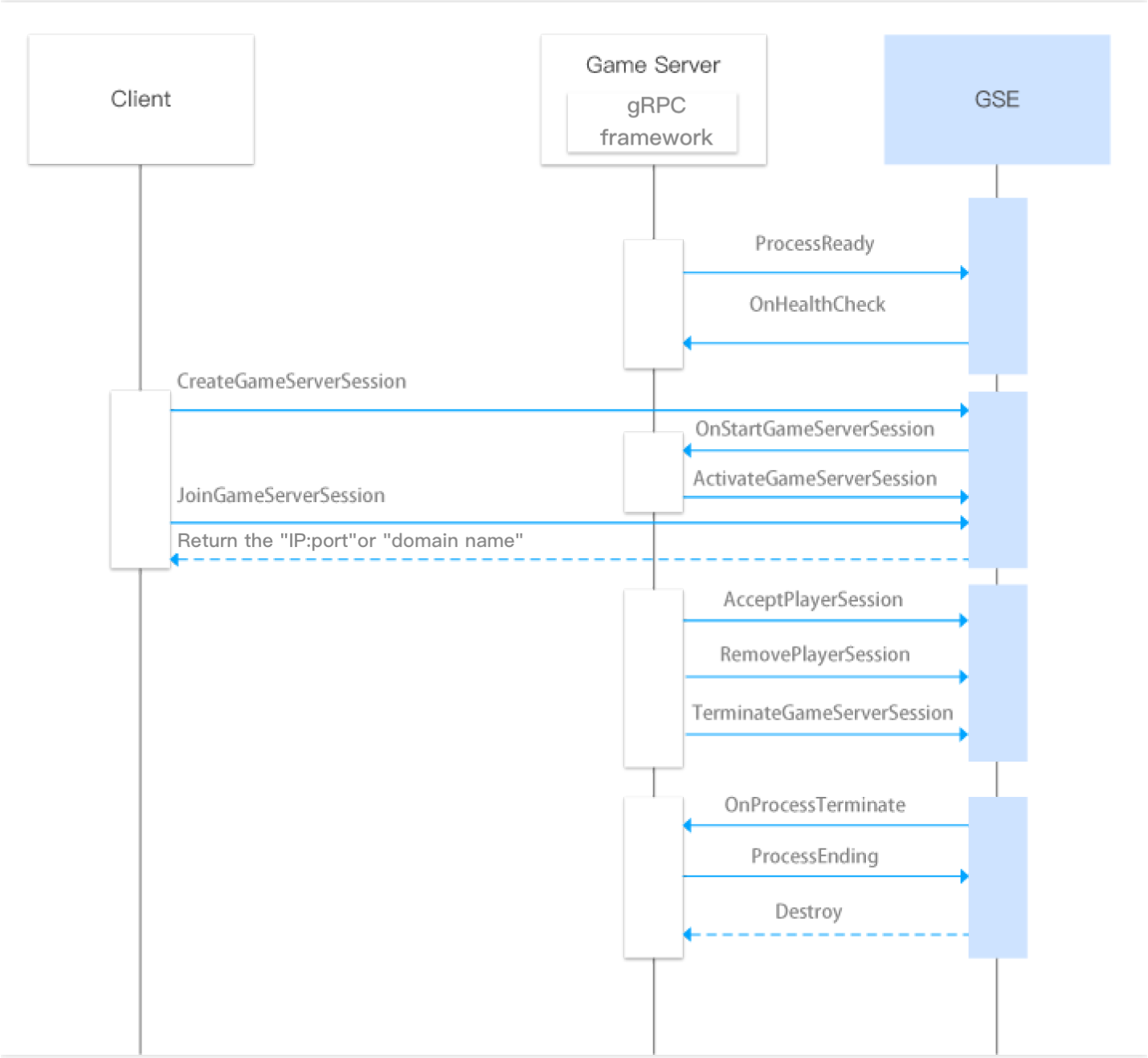
ii. Generate the `pb.cc` and `pb.h` files.

```
protoc --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` *.proto
```

iii. Generate the corresponding gRPC code.

iv. Move the eight generated files to an appropriate location in the project.

Game Process Integration Process



Game server callback API list

API Name	API Description
OnHealthCheck	Runs health check
OnStartGameServerSession	Receives game server session
OnProcessTerminate	Ends game process

Game server active API list

API Name	API Description
ProcessReady	Gets process ready
ActivateGameServerSession	Activates game server session
AcceptPlayerSession	Receives player session
RemovePlayerSession	Removes player session
DescribePlayerSessions	Gets player session list
UpdatePlayerSessionCreationPolicy	Updates player session creation policy
TerminateGameServerSession	Ends game server session
ProcessEnding	Ends process
ReportCustomData	Reports custom data

Others

When the game process uses gRPC to call a game server active API, you need to add two fields to `meta` of the gRPC request.

Field	Description	Type
pid	`pid` of the current game process	string
requestId	`requestId` of the current request, which is used to uniquely identify a request	string

1. Generally, after the server is initialized, the process will check itself to see whether it can provide services, and the game server will call the `ProcessReady` API to notify GSE that the process is ready to host a game server session. After receiving the notification, GSE will change the status of the server instance to "Active".

```
static bool luaProcessReady(std::vector <std::string> &logPath, int clientPort, int grpcPort)
{
GseResponse reply;
// Log path. Set the ports.
Status status = GGseManager->ProcessReady(logPath, clientPort, grpcPort, reply);
// Ready to provide services
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
```

```
}
return true;
}
```

2. After the process is ready, GSE will call the `OnHealthCheck` API to perform a health check on the game server every minute. If the health check fails three consecutive times, the process will be considered to be unhealthy, and no game server sessions will be assigned to it.

```
Status GameServerGrpcSdkServiceImpl::OnHealthCheck(ServerContext* context, const HealthCheckRe
quest* request, HealthCheckResponse* reply)
{
healthStatus = GSESDK()->exec("return OnHealthCheck()");
std::cout << "healthStatus=" << healthStatus << std::endl;
reply->set_healthstatus(healthStatus);
return Status::OK;
}
```

3. Because the client calls the CreateGameServerSession API to create a game server session and assigns it to a process, GSE will be triggered to call the `onStartGameServerSession` API for the process and change the status of `GameServerSession` to "Activating".

```
Status GameServerGrpcSdkServiceImpl::OnStartGameServerSession(ServerContext* context, const St
artGameServerSessionRequest* request, GseResponse* reply)
{
auto gameServerSession = request->gameserversession();
GGseManager->SetGameServerSession(gameServerSession);
std::ostringstream o;
o << "return OnStartGameServerSession('" << gameServerSession.gameserversessionid() << "'," <<
gameServerSession.maxplayers() << ")";
std::string luaCmd = o.str();
bool res = GSESDK()->exec(luaCmd);
return Status::OK;
}
```

4. After the game server receives `onStartGameServerSession` , you need to handle the logic or resource allocation by yourself. After everything is ready, the game server will call the `ActivateGameServerSession` API to notify GSE that the game server session has been assigned to a process and is ready to receive player requests and will change the server status to "Active".

```
static bool luaActivateGameServerSession(const std::string &gameServerSessionId, int maxPlayer
s) {
```

```
GseResponse reply;
Status status = GGseManager->ActivateGameServerSession(gameServerSessionId, maxPlayers, repl
y);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

5. After the client calls the JoinGameServerSession API for the player to join, the game server will call the `AcceptPlayerSession` API to verify the validity of the player. If the connection is accepted, the status of `PlayerSession` will be set to "Active". If the client receives no response within 60 seconds after calling the `JoinGameServerSession` API, it will change the status of `PlayerSession` to "Timeout" and then call `JoinGameServerSession` again.

```
static bool luaAcceptPlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->AcceptPlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

6. After the game ends or the player exits, the game server will call the `RemovePlayerSession` API to remove the player, change the status of `playersession` to "Completed", and reserve the player slot in the game server session.

```
static bool luaRemovePlayerSession(const std::string &gameServerSessionId, const std::string &
playerSessionId) {
GseResponse reply;
Status status = GGseManager->RemovePlayerSession(gameServerSessionId, playerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

7. After a game server session (a game battle or a service) ends, the game server will call the `TerminateGameServerSession` API to end the `GameServerSession` and change its status to `Terminated`.

```
static bool luaTerminateGameServerSession(const std::string &gameServerSessionId) {
GseResponse reply;
Status status = GGseManager->TerminateGameServerSession(gameServerSessionId, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

8. In case of health check failure or reduction, GSE will call the `OnProcessTerminate` API to end the game process. The reduction will be triggered according to the protection policy configured in the GSE console.

```
Status GameServerGrpcSdkServiceImpl::OnProcessTerminate(ServerContext* context, const ProcessT
erminateRequest* request, GseResponse* reply)
{
auto terminationTime = request->terminationtime();
std::to_string(terminationTime));
std::ostringstream o;
o << "OnProcessTerminate(" << terminationTime << ")";
std::string luaCmd = o.str();
GSESDK()->execWithNilResult(luaCmd);
return Status::OK;
}
```

9. The game server calls the `ProcessEnding` API to end the process immediately, change the server process status to "Terminated", and repossess the resources.

```
// Active call: a game battle corresponds to a process. The `ProcessEnding` API will be active
ly called after the game battle ends
// Passive call: in case of reduction, process exception, or health check failure, the `Proces
sEnding` API will be called passively according to the protection policy. If a full protection
or time-period protection policy is configured, it is required to determine whether there are
any players in the game server session before the passive call can be made
static bool luaProcessEnding() {
GseResponse reply;
Status status = GGseManager->ProcessEnding(reply);
GSESDK()->setReplyStatus(status);
```

```
if (!status.ok()) {
return false;
}
return true;
}
```

0. The game server calls the `DescribePlayerSessions` API to get the information of the player in the game server session (which is optional based on your actual business needs).

```
static bool luaDescribePlayerSessions(const std::string &gameServerSessionId,
const std::string &playerId,
const std::string &playerSessionId,
const std::string &playerSessionStatusFilter, const std::string &nextToken,
int limit) {
DescribePlayerSessionsResponse reply;
Status status = GGseManager->DescribePlayerSessions(gameServerSessionId,playerId, playerSessio
nId, playerSessionStatusFilter, nextToken, limit, reply);
GSESDK()->setDescribePlayerSessionsResponse(reply);
if (!status.ok()) {
return false;
}
return true;
}
```

1. The game server calls the `UpdatePlayerSessionCreationPolicy` API to update the player session creation policy and set whether to accept new players, i.e., whether to allow new players to join a game session (which is optional based on your actual business needs).

```
static bool luaUpdatePlayerSessionCreationPolicy(const std::string &newpolicy) {
GseResponse reply;
Status status = GGseManager->UpdatePlayerSessionCreationPolicy(newpolicy, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

2. The game server calls the `ReportCustomData` API to notify GSE of the custom data (which is optional based on your actual business needs).

```
static bool luaReportCustomData(int currentCustomCount, int maxCustomCount) {
GseResponse reply;
Status status = GGseManager->ReportCustomData(currentCustomCount, maxCustomCount, reply);
GSESDK()->setReplyStatus(status);
if (!status.ok()) {
return false;
}
return true;
}
```

Launching Server for GSE to Call

Server running: launch `GrpcServer` .

```
// Launch the gRPC server
std::thread tGrpc(&GameServerGrpcSdkServiceImpl::StartGrpcServer, GGameServerGrpcSdkService);
sem_wait(&(GGameServerGrpcSdkService->sem));
auto grpcPort = GGameServerGrpcSdkService->GetGrpcPort();
```

Connecting Client to gRPC Server of GSE

Server connecting: create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.

```
void GseManager::InitStub() {
auto channel = grpc::CreateChannel("127.0.0.1:5758", grpc::InsecureChannelCredentials());
stub_ = GseGrpcSdkService::NewStub(channel);
}
```

Demo for Lua

1. Click here to download the code of the Demo for Lua.
2. Generate the gRPC code.
   The Demo for Lua relies on the C++ framework, with gRPC code generated in the `cpp-demo/source/grpcsdk` directory, so there is no need to generate it again.
3. Launch the server for GSE to call.

- Implement the server.
  `grpcserver.cpp` in the `lua-demo/source/api` directory implements three server APIs.
- Run the server.
  `main.cpp` in the `lua-demo` directory launches `GrpcServer` .

4. Connect the client to the gRPC server of GSE.
   - Implement the client.
     `GSESdkHandleWrapper.cpp` in the `lua-demo/source/lua` directory implements nine client APIs.
   - Connect to the server.
     Create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.
5. Compile and run the project.
   i. Install CMake.
   - Install GCC v4.9 or above.
   - Install the LuaJIT and Boost development kits:

```
yum install -y luajit-devel
yum install -y boost-devel
yum install -y cmake
```

   - Download the code and run the following command in the `lua-demo` directory:

```
mkdir build
cd build
cmake ..
make
cp ../source/lua/gse.lua .
```

     The corresponding `lua-demo` executable file will be generated. Run `./lua-demo` to launch it.
   - Package the executable file `lua-demo.cpp` as an asset package and configure the launch path as `lua-demo` with no launch parameter needed.
   - Create a server fleet and deploy the asset package on it. After that, you can perform various operations such as scaling.

gRPC Unity Tutorial

Last updated : 2021-08-30 15:52:50

This document describes how to integrate Unity with GSE SDK. The overall process mainly consists of two tasks:

1. Integrating Unity with gRPC
2. Integrating Unity with GSE SDK

Prerequisites

You have already installed Unity Hub and Unity IDE.

> Note :
> This document uses 2018.3.5f1 or 2019.4.9f1 Unity engine and MacOS as an example.

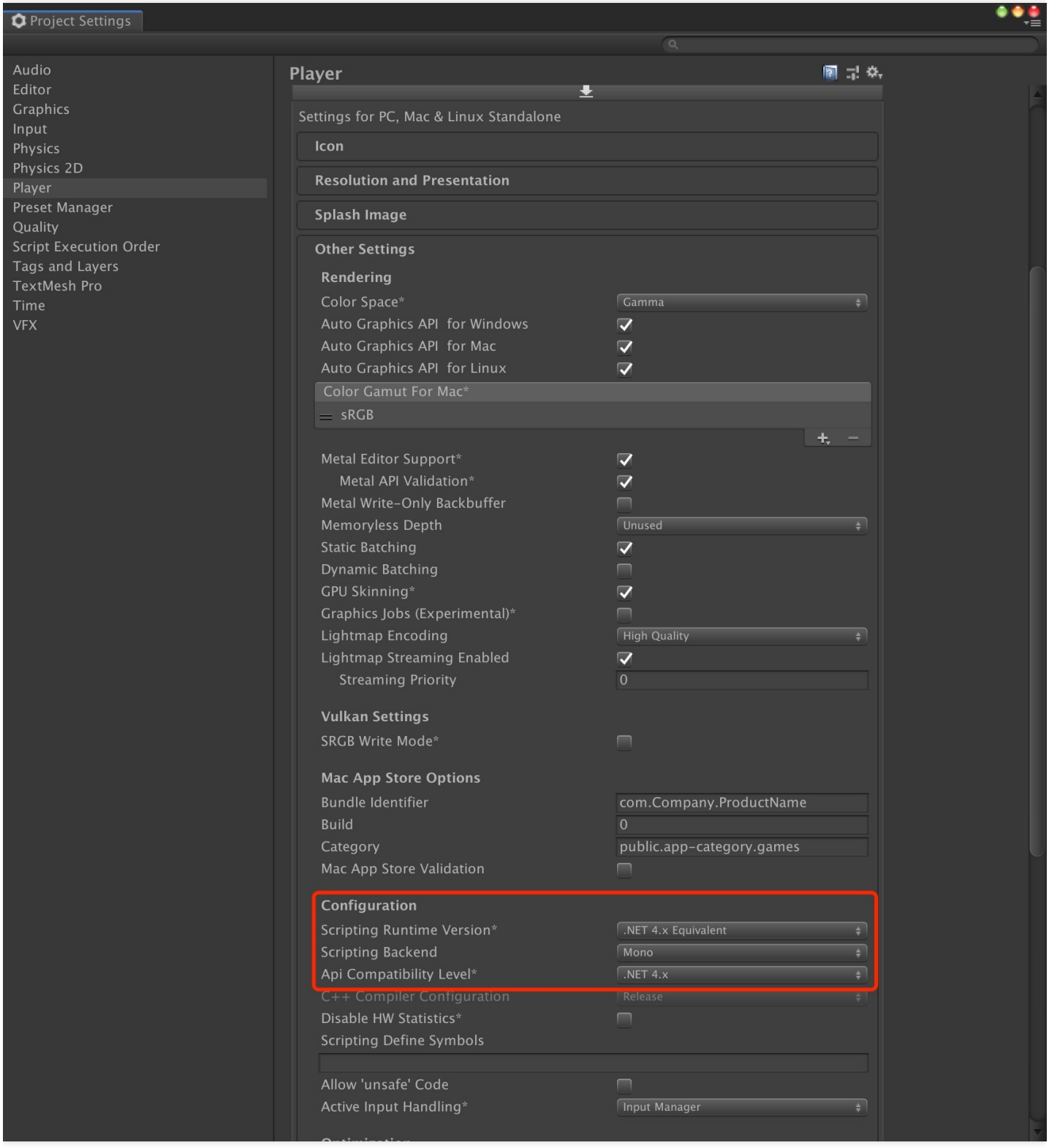
Integrating Unity with gRPC

gRPC has experimental support for Unity. For more information, see README. Perform the following steps to integrate Unity with gRPC:

Step 1: create a Unity project

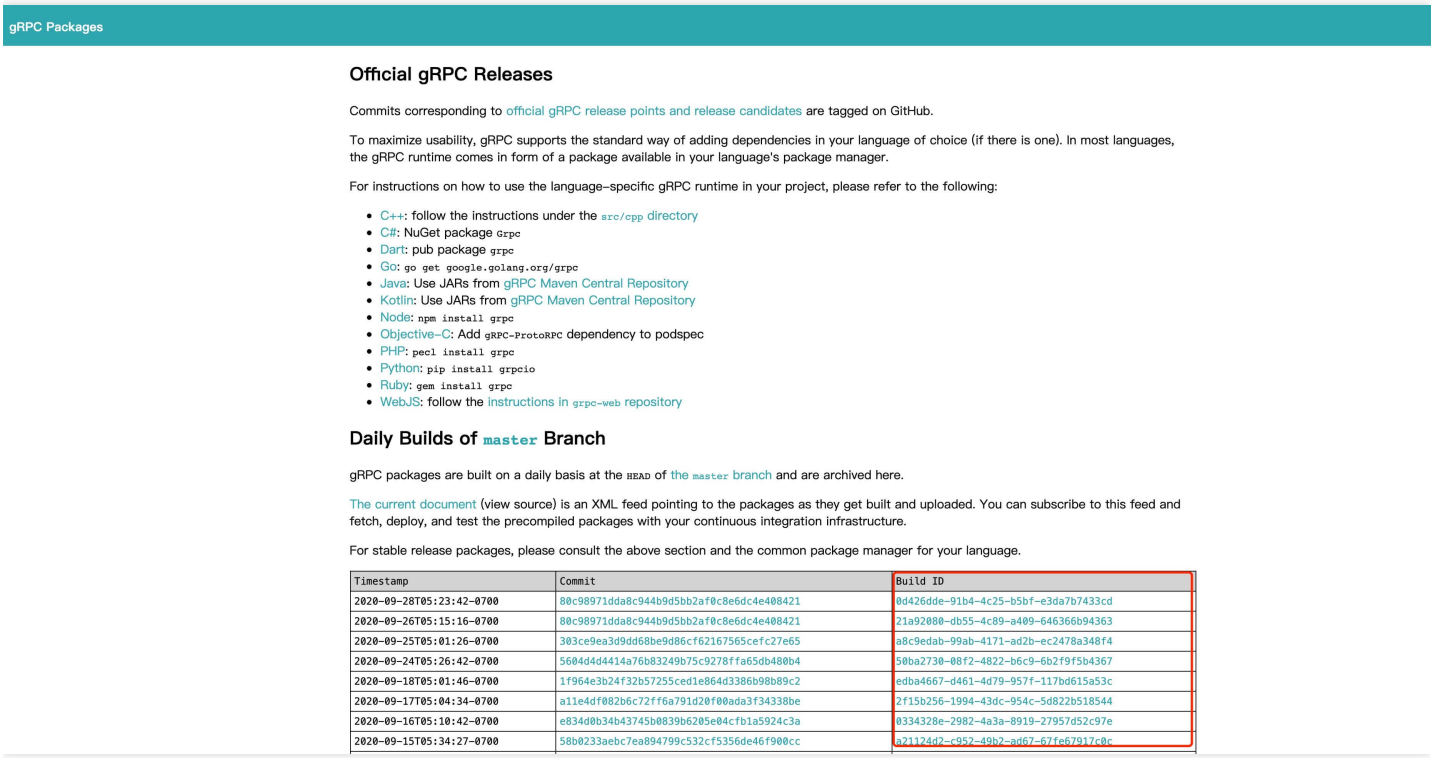
Because gRPC APIs are only available for `.NET 4.5+`, it is necessary to create a Unity project equivalent to `.NET 4.x` at **Edit** > **Project Setting** > **Player** > **Configuration** > **Scripting**

Runtime Version.



Step 2: download grpc_unity_package

Download the latest development version of `grpc_unity_package.VERSION.zip` here. Click `Buidld ID` to redirect to the download page.



gRPC Packages

Official gRPC Releases

Commits corresponding to official gRPC release points and release candidates are tagged on GitHub.

To maximize usability, gRPC supports the standard way of adding dependencies in your language of choice (if there is one). In most languages, the gRPC runtime comes in form of a package available in your language's package manager.

For instructions on how to use the language-specific gRPC runtime in your project, please refer to the following:

- C++: follow the instructions under the `src/cpp` directory
- C#: NuGet package `Grpc`
- Dart: pub package `grpc`
- Go: `go get google.golang.org/grpc`
- Java: Use JARs from gRPC Maven Central Repository
- Kotlin: Use JARs from gRPC Maven Central Repository
- Node: `npm install grpc`
- Objective-C: Add `gRPC-ProtoRPC` dependency to podspec
- PHP: `pecl install grpc`
- Python: `pip install grpcio`
- Ruby: `gem install grpc`
- WebJS: follow the instructions in `grpc-web` repository

Daily Builds of `master` Branch

gRPC packages are built on a daily basis at the `HEAD` of the `master` branch and are archived here.

The current document (view source) is an XML feed pointing to the packages as they get built and uploaded. You can subscribe to this feed and fetch, deploy, and test the precompiled packages with your continuous integration infrastructure.

For stable release packages, please consult the above section and the common package manager for your language.

Timestamp	Commit	Build ID
2020-09-28T05:23:42-0700	80c98971dda8c944b9d5bb2af0c8e6dc4e408421	0d426dde-91b4-4c25-b5bf-e3da7b7433cd
2020-09-26T05:15:16-0700	80c98971dda8c944b9d5bb2af0c8e6dc4e408421	21a92080-db55-4c89-a409-646366b94363
2020-09-25T05:01:26-0700	303ce9ea3d9dd68be9d86cf62167565cefc27e65	a8c9edab-99ab-4171-ad2b-ec2478a348f4
2020-09-24T05:26:42-0700	5604d4d4414a76b83249b75c9278ffa65db480b4	50ba2730-08f2-4822-b6c9-6b2f9f5b4367
2020-09-18T05:01:46-0700	1f964e3b24f32b57255ced1e864d3386b98b89c2	edba4667-d461-4d79-957f-117bd615a53c
2020-09-17T05:04:34-0700	a11e4df082b6c72ff6a791d20f00ada3f34338be	2f15b256-1994-43dc-954c-5d822b518544
2020-09-16T05:10:42-0700	e834d0b34b43745b0839b6205e04cfb1a5924c3a	0334328e-2982-4a3a-8919-27957d52c97e
2020-09-15T05:34:27-0700	58b0233aebc7ea894799c532cf5356de46f900cc	a21124d2-c952-49b2-ad67-67fe67917c0c

Click to download `grpc_unity_package.VERSION.zip` under the `c#` directory.

Build: `0d426dde-91b4-4c25-b5bf-e3da7b7433cd` [invocation]
Timestamp: **2020-09-28T05:23:42-0700**
Branch: `master`
Commit: `80c98971dda8c944b9d5bb2af0c8e6dc4e408421`

gRPC `protoc` Plugins

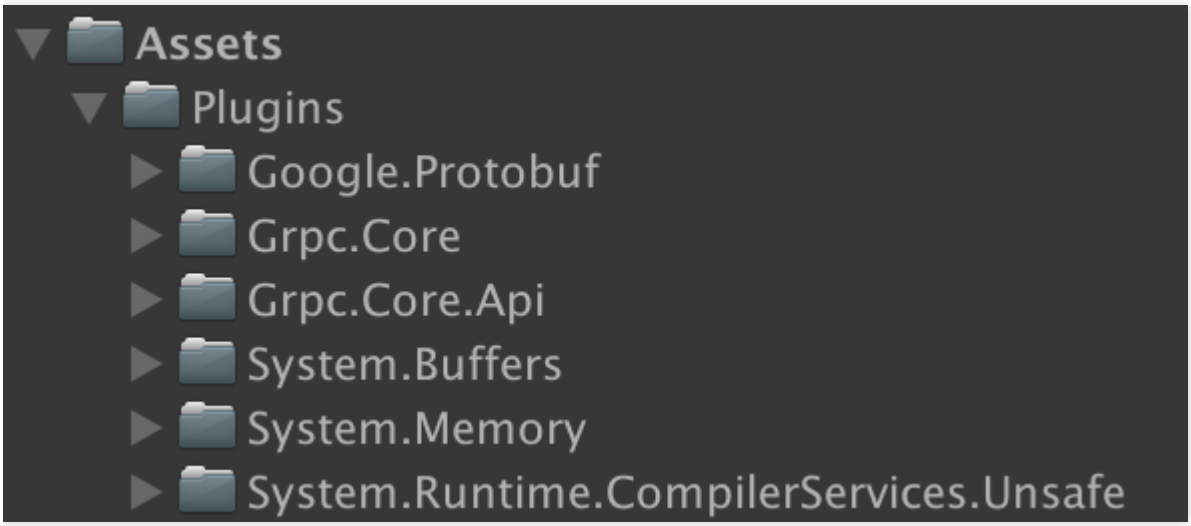
grpc-protoc_linux_x64-1.33.0-dev.tar.gz	c4b4a354363613bf011e85a4635ed6a193f710f849df8c109c093d7c022d5f72
grpc-protoc_linux_x86-1.33.0-dev.tar.gz	1fa6c3f6afb5cca9e5a70ff920324184b33302a48d61a259cf796434e34deaea
grpc-protoc_macos_x64-1.33.0-dev.tar.gz	e527428c70a8cd081ce515c6bbd045b787c61750c55e7d21e9555c0a181819fa
grpc-protoc_macos_x86-1.33.0-dev.tar.gz	fa13b3a80b5a0ab8966cd80203145294 70f41788f7db18508005a01132578306
grpc-protoc_windows_x64-1.33.0-dev.zip	8dcdcb9563dd8dd0762c24aefabab56e197725eda181c3e2551df30d2e9d0eaa
grpc-protoc_windows_x86-1.33.0-dev.zip	9ca0e1f8e557545f4a873b791fd1e70d403add85fa2571fd666109c6da66cb02

C#

grpc_unity_package.2.33.0-dev.zip	18b5313758b2b7451e41d37bfc12992e6b61b055017a11b33b7a953b7e58b1a8
Grpc.2.33.0-dev202009281202.nupkg	01ca8739a97fbab5536036f16f2e30f17a0c1d9b409ee719b2991d58b6b929ac
Grpc.Auth.2.33.0-dev202009281202.nupkg	8d0b7294a82b093739f470cfab636bbfe08e62ed3177d463033acc75420793e4
Grpc.Core.2.33.0-dev202009281202.nupkg	57739eac5f4b995e16a570b202b45e10c315a9e85f036882b8c8a47294ce1409
Grpc.Core.Api.2.33.0-dev202009281202.nupkg	95ae31673348685bcfa11005123fd68fe33ee5428224c2090f4f407187ee8597
Grpc.Core.NativeDebug.2.33.0-dev202009281202.nupkg	824e20d42d3066e3ceaa17b3240f5ef8a605eac3258dff2c42f3ba6619489260
Grpc.Core.Testing.2.33.0-dev202009281202.nupkg	3bb4565eee60f6bc433107e62df5beff6442eafdb1a2d5742e46b9597cc898f1
Grpc.HealthCheck.2.33.0-dev202009281202.nupkg	546ac6a534496e67f0eac49a583feff0389fbb74225af93d88fc13d0256dab21
Grpc.Reflection.2.33.0-dev202009281202.nupkg	5f6f7ba895e1da7e6ddce02340d95bce0ce0133853b9ff28c5636f33b20fff87
Grpc.Tools.2.33.0-dev202009281202.nupkg	4a70f6bbf67a8b39a6470aa902ff2d424dee893b11a602a83a964d1de367443d

Step 3: decompress the package

Decompress the downloaded `.zip` package to the `Assets` directory of the Unity project, as shown below:



Step 4: test the package

Unity Editor will fetch files and automatically add them to the project for your use of gRPC and Protobuf in codes. If Unity Editor prompts an error, see FAQs for troubleshooting.

Integrating Unity with GSE SDK

Complete the following steps to integrate Unity with GSE SDK:

Step 1: obtain the GSE SDK Protobuf files

Obtain the `GameServerGrpcSdkService.proto` and `GseGrpcSdkService.proto` files of GSE SDK Protobuf. For more information, see proto File

Step 2: generate C# codes based on Protobuf

1. Access the grpc_unity_package.VERSION.zip page again to download the gRPC protoc Plugin package compatible with your operating system.

Build: edd81ac6-e3d1-461a-a263-2b06ae913c3f [invocation]
Timestamp: 2019-12-02T03:56:08-0800
Branch: master
Commit: a02d6b9be81cbadb60eed88b3b44498ba27bcba9

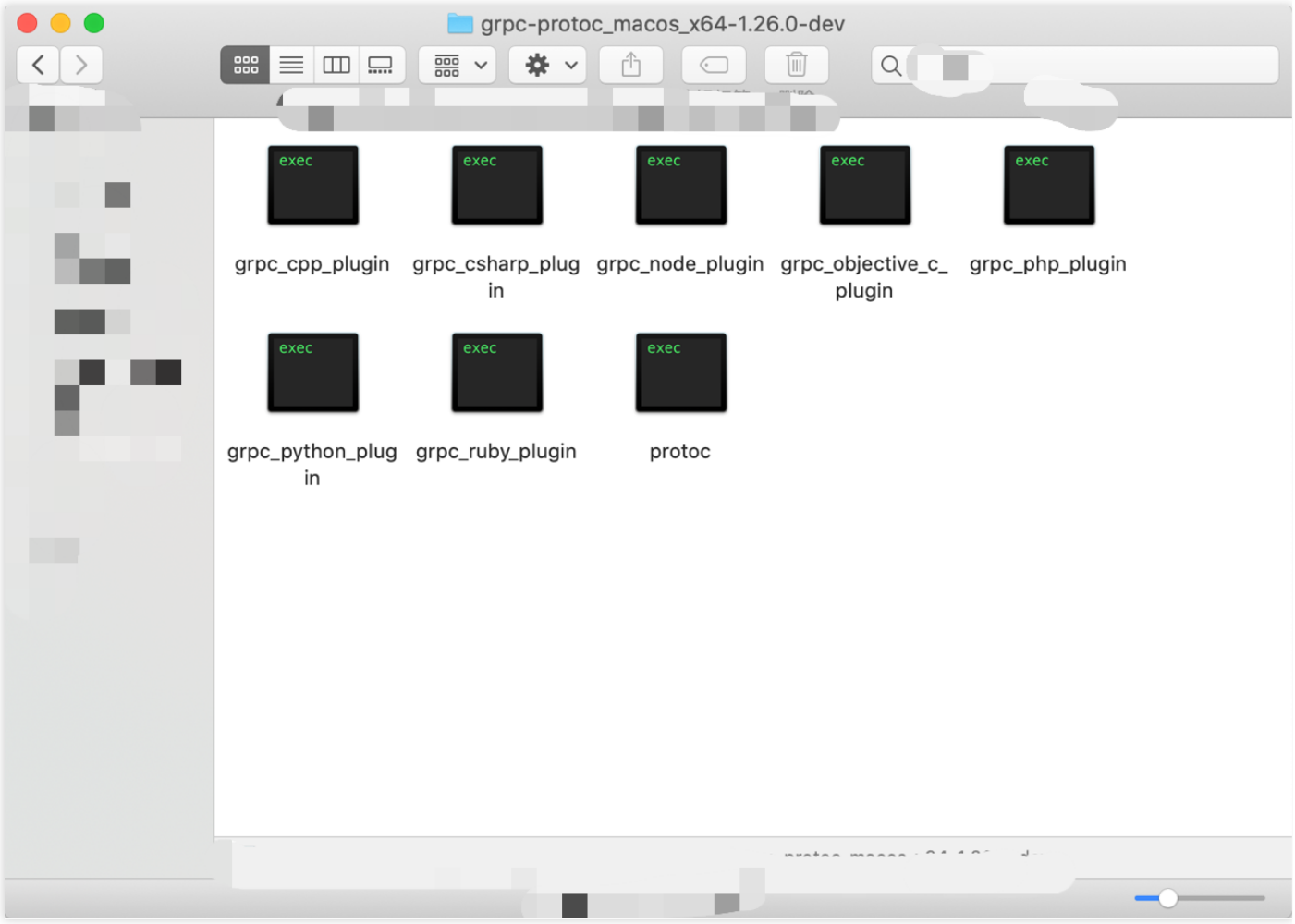
gRPC protoc Plugins

grpc-protoc_linux_x64-1.26.0-dev.tar.gz	534030f048c54b6d1c4788a7694a92fde11b36f1ef6386fe71fbc5ea6be73117
grpc-protoc_linux_x86-1.26.0-dev.tar.gz	8be18206322c8702fb3d3fd631f9a5218495bee856a7f65cb2693923175d550d
grpc-protoc_macos_x64-1.26.0-dev.tar.gz	51a1f7ba55dacd7f9ee7ddd5b445433b4d9ff9ca0166a08e99573827a379165e
grpc-protoc_macos_x86-1.26.0-dev.tar.gz	ac3419dea1589e367e6245cfd68489890928d6250de15b3d591ff48d3e9aecbc
grpc-protoc_windows_x64-1.26.0-dev.zip	8a76893e05a8d838572c6c4ff0be877029f50b6cc9de93cc3e9b045040784fdd
grpc-protoc_windows_x86-1.26.0-dev.zip	2b552c652d97cee13bd7356a38c642c8578b4c26444c20c09cd9c8e37faecb51

C#

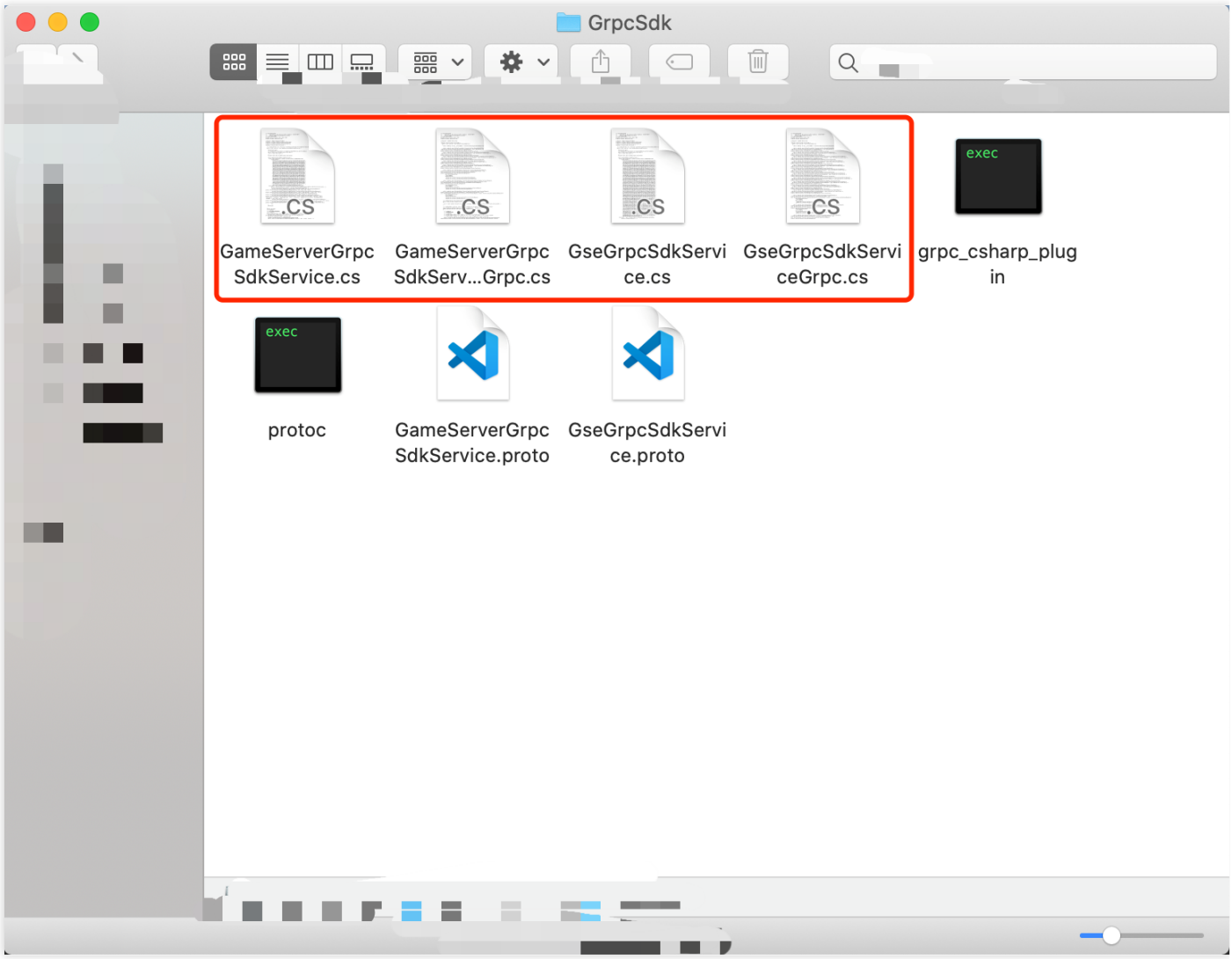
grpc_unity_package.2.26.0-dev.zip	509af7278e725cc6a291d354365db930ea1bc96fe53bd96dc88cb33ed1c3e797
Grpc.2.26.0-dev201912021138.nupkg	5ac6ef42a6dbb17f15073abfc986764a18872437f0f357c6a17abfc93aaf20f7
Grpc.Auth.2.26.0-dev201912021138.nupkg	d0e2bb6538478ced3319d962dd1ac4519f0ab93ee7b16b87cafc903cebbda4c3
Grpc.Core.2.26.0-dev201912021138.nupkg	5efd93d0519ec9ccb91f04d0a6fa0f7c596fc7b40fe2f1506785c5de008de1fb
Grpc.Core.Api.2.26.0-dev201912021138.nupkg	822baa7faef5caa6b04666f8bd926c3d316e245f4960f351887f034df4a53541
Grpc.Core.NativeDebug.2.26.0-dev201912021138.nupkg	1211e8bd5336b612c866d84dc85bb32257a7907796da49b0f827000a7bbdd73e
Grpc.Core.Testing.2.26.0-dev201912021138.nupkg	4e5faa71af895c476418376fab48c48f65ac73d469049d490ec5dd4674201a91
Grpc.HealthCheck.2.26.0-dev201912021138.nupkg	a07b5e1399fcf8dcf94b0ca90f96bfdf84d582d69a7ddd03ec81baa79c460ef3
Grpc.Reflection.2.26.0-dev201912021138.nupkg	823c4dbece26be83a210c778b95bfa07e43d0a439851d67a247cbd048cea009c
Grpc.Tools.2.26.0-dev201912021138.nupkg	adb038ceac8b820f2875d66118775a42b2055fb9a0bb77fef3535a80cc56bb5e

2. Decompress the package to obtain the `protoc` and `grpc_csharp_plugin` executable programs.



3. Copy `protoc` and `grpc_csharp_plugin` executable programs to the same directory as the Protobuf file. Run the following two commands according to the operating system to generate `C#` codes:
   - **For MAC and Linux OS:**
     - `protoc -I ./ --csharp_out=. GseGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin`
     - `protoc -I ./ --csharp_out=. GameServerGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin`
   - **For Windows OS:**
     - `./protoc -I ./ --csharp_out=. GseGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin.exe`
     - `./protoc -I ./ --csharp_out=. GameServerGrpcSdkService.proto --grpc_out=. --plugin=protoc-gen-grpc=grpc_csharp_plugin.exe`

Four `.cs` code files are generated as shown in the following figure.



Step 3: develop and use GSE SDK on the Unity server

Copy the four `.cs` files generated in the Step 2 to the Unity project (to a separate folder under the `Assets/Scripts/` directory) and use GSE SDK for the development. For more information, see Unity DEMO.

1. Implement the `OnHealthCheck`, `OnStartGameServerSession` and `OnProcessTerminate` APIs defined by `gameserver_grpcsdk_service.proto`.

```
public class GrpcServer : GameServerGrpcSdkService.GameServerGrpcSdkServiceBase
{
private static Logs logger
{
get
```



```
{
return new Logs();
}
}
// Health checks
public override Task<HealthCheckResponse> OnHealthCheck(HealthCheckRequest request, ServerCall
Context context)
{
logger.Println($"OnHealthCheck, HealthStatus: {GseManager.HealthStatus}");
logger.Println($"OnHealthCheck, GameServerSession: {GseManager.GetGameServerSession()}");
return Task.FromResult(new HealthCheckResponse
{
HealthStatus = GseManager.HealthStatus
});
}
// Receive game sessions
public override Task<GseResponse> OnStartGameServerSession(StartGameServerSessionRequest reque
st, ServerCallContext context)
{
logger.Println($"OnStartGameServerSession, request: {request}");
GseManager.SetGameServerSession(request.GameServerSession);
var resp = GseManager.ActivateGameServerSession(request.GameServerSession.GameServerSessionId,
request.GameServerSession.MaxPlayers);
logger.Println($"OnStartGameServerSession, resp: {resp}");
return Task.FromResult(resp);
}
// End the game process
public override Task<GseResponse> OnProcessTerminate(ProcessTerminateRequest request, ServerCa
llContext context)
{
logger.Println($"OnProcessTerminate, request: {request}");
// Set the process termination time
GseManager.SetTerminationTime(request.TerminationTime);
// Terminate game server sessions
GseManager.TerminateGameServerSession();
// Exit the process
GseManager.ProcessEnding();
return Task.FromResult(new GseResponse());
}
}
```

2. Develop Unity server programs (taking ChatServer as an example).

```
public static void StartChatServer(int clientPort)
{
RegisterHandlers();
```

```
logger.Println("ChatServer Listen at " + clientPort);
NetworkServer.Listen(clientPort);
}
```

3. Develop the gRPC server.

```
public static void StartGrpcServer(int clientPort, int grpcPort, string logPath)
{
try
{
Server server = new Server
{
Services = { GameServerGrpcSdkService.BindService(new GrpcServer()) },
Ports = { new ServerPort("127.0.0.1", grpcPort, ServerCredentials.Insecure) },
};
server.Start();
logger.Println("GrpcServer Start On localhost:" + grpcPort);
GseManager.ProcessReady(new string[] { logPath }, clientPort, grpcPort);
}
catch (System.Exception e)
{
logger.Println("error: " + e.Message);
}
}
```

4. Launch the implemented server and the gRPC server.

```
public class StartServers : MonoBehaviour
{
private int grpcPort = PortServer.GenerateRandomPort(2000, 6000);
private int chatPort = PortServer.GenerateRandomPort(6001, 10000);
private const string logPath = "./log/log.txt";
// Start is called before the first frame update
[Obsolete]
void Start()
{
// Start ChatServer By UNet's NetWorkServer, Listen on UDP protocol
MyChatServer.StartChatServer(chatPort);
// Start GrpcServer By Grpc, Listen on TCP protocol
MyGrpcServer.StartGrpcServer(chatPort, grpcPort, logPath);
}
[Obsolete]
void OnGUI()
{
```

```
}
}
```

Unity DEMO

1. Click here to download the code of the Demo for Unity.
2. Import grpc unity package.
   Decompress grpc_unity_package in the Step 2 to the `unity-demo/Assets` directory of the Demo project.
3. Generate C# codes based on the Protobuf file.
4. Launch the server for GSE to call.
   - Implement the server: implement the three server APIs in the `GrpcServer.cs` file under the `unity-demo/Assets/Scripts/Api` directory.
   - Run the server: create `gRPC Server` and `StartServers.cs` in the `MyGrpcServer.cs` file under the `unity-demo/Assets/Scripts` directory to launch `gRPC Server` .
5. Connect the client to the gRPC server of GSE.
   - Implement the client: implement the nine client APIs in the `Gsemanager.cs` file under the `unity-demo/Assets/Scripts/Gsemanager` directory.
   - Connect to the server: create a gRPC channel, specify the host name and server port to connect to, and use this channel to create a stub instance.
6. Compile and run the program
   Use Unity Editor to encapsulate the executable program of the target system into an asset package, and configure the actual name of the executable program at the launch path.

Getting Server Address TencentCloud API Calling Method

Last updated : 2021-03-30 10:13:29

A client API is provided as a TencentCloud API and can be called in the following ways:

1. SDK Call

You can use Tencent Cloud Software Development Kit (SDK) v3.0 to call a client TencentCloud API. The SDK supports various programming languages such as PHP, Python, Java, Go, .NET, Node.js, and C++.

Note :

Currently, GSE supports SDK v3.0. For detailed directions, please see the SDK overview.

2. Online Debugging

You can use API Explorer to call a client TencentCloud API. This tool provides various capabilities such as online call, signature verification, SDK code generation, and quick API search.

Note :

In API 3.0 Explorer, select "GSE" and then select a TencentCloud API under "Console APIs" or "Service Management APIs" for online debugging.

3. Direct Encapsulation

You can use the HTTP request method of a domain name or an API name to call a client TencentCloud API.

Note :

TencentCloud APIs of GSE have been upgraded to v3.0. For detailed directions, please see TencentCloud API calling methods.

Creating Game Server Session

Last updated : 2020-07-27 10:26:38

Overview

- You can use a client TencentCloud API to create a game server session in the following two ways:
  - Create in a server fleet to implement auto scaling and health check.
  - Create through an alias to implement zero downtime update.
- One game server session is placed in one server process, but the client API calling process varies by supporting mode of the game server session.

Client API Calling Process

One game server session supports one game

If one game server session supports only one game, you can call a client API in the following steps:

1. Create a game server session through a server fleet or alias. For detailed directions, please see the API document CreateGameServerSession.

> **Note :**
>
> The following sample code is based on Java:

```
public class CreateGameServerSession
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);
```

```
String params = "{}";
CreateGameServerSessionRequest req = CreateGameServerSessionRequest.fromJsonString(params, Cre
ateGameServerSessionRequest.class);

CreateGameServerSessionResponse resp = client.CreateGameServerSession(req);

System.out.println(CreateGameServerSessionRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

2. Join the created game server session. For detailed directions, please see the API document JoinGameServerSession.

```
public class JoinGameServerSession
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
JoinGameServerSessionRequest req = JoinGameServerSessionRequest.fromJsonString(params, JoinGameSe
rverSessionRequest.class);

JoinGameServerSessionResponse resp = client.JoinGameServerSession(req);

System.out.println(JoinGameServerSessionRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

One game server session supports multiple games or one service

If one game server session supports multiple games or one service (such as login), you can all a client API in the following steps:

1. Query the game server session list to check whether there is any game server session. For detailed directions, please see the API document DescribeGameServerSessions.

```
public class DescribeGameServerSessions
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
DescribeGameServerSessionsRequest req = DescribeGameServerSessionsRequest.fromJsonString(param
s, DescribeGameServerSessionsRequest.class);

DescribeGameServerSessionsResponse resp = client.DescribeGameServerSessions(req);

System.out.println(DescribeGameServerSessionsRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

You can also search for existing sessions in the game server session list. For detailed directions, please see the API document SearchGameServerSessions.

```
public class SearchGameServerSessions
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");
```

```
HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
SearchGameServerSessionsRequest req = SearchGameServerSessionsRequest.fromJsonString(params, S
earchGameServerSessionsRequest.class);

SearchGameServerSessionsResponse resp = client.SearchGameServerSessions(req);

System.out.println(SearchGameServerSessionsRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

2. If a game server session exists, you can directly join it. For detailed directions, please see the API document JoinGameServerSession or the sample code in this document.

3. If no game server sessions exist, you need to create one first. For detailed directions, please see the API document CreateGameServerSession or the sample code in this document. Then, join the created session. For detailed directions, please see the API document JoinGameServerSession or the sample code in this document.

Note :

You can use API 3.0 Explorer for online debugging. You can select TencentCloud APIs under "Game Server Engine" > "Service Management APIs" on the left sidebar and perform operations such as "Code Generation" and "Online Call".

Placing Game Server Session

Last updated : 2021-04-20 15:06:54

Overview

You can use a client TencentCloud API to place a game server session, that is, you can implement nearby resource scheduling and cross-region disaster recovery through a game server queue.

Client API Calling Process

1. First, check whether a game server session has been placed in a process. For detailed directions, please see the API document DescribeGameServerSessionPlacement.

> **Note :**
>
> The following sample code is based on Java:

```
public class DescribeGameServerSessionPlacement
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
DescribeGameServerSessionPlacementRequest req = DescribeGameServerSessionPlacementRequest.fromJso
nString(params, DescribeGameServerSessionPlacementRequest.class);

DescribeGameServerSessionPlacementResponse resp = client.DescribeGameServerSessionPlacement(req);

System.out.println(DescribeGameServerSessionPlacementRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
```

```
System.out.println(e.toString());
}
}
}
```

2. Start placing the game server session. For detailed directions, please see the API document StartGameServerSessionPlacement.

```
public class StartGameServerSessionPlacement
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");

HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
StartGameServerSessionPlacementRequest req = StartGameServerSessionPlacementRequest.fromJsonStrin
g(params, StartGameServerSessionPlacementRequest.class);

StartGameServerSessionPlacementResponse resp = client.StartGameServerSessionPlacement(req);

System.out.println(StartGameServerSessionPlacementRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

3. Stop placing the game server session. For detailed directions, please see the API document StopGameServerSessionPlacement.

```
public class StopGameServerSessionPlacement
{
public static void main(String [] args) {
try{

Credential cred = new Credential("", "");
```

```
HttpProfile httpProfile = new HttpProfile();
httpProfile.setEndpoint("gse.tencentcloudapi.com");

ClientProfile clientProfile = new ClientProfile();
clientProfile.setHttpProfile(httpProfile);

GseClient client = new GseClient(cred, "", clientProfile);

String params = "{}";
StopGameServerSessionPlacementRequest req = StopGameServerSessionPlacementRequest.fromJsonString(
params, StopGameServerSessionPlacementRequest.class);

StopGameServerSessionPlacementResponse resp = client.StopGameServerSessionPlacement(req);

System.out.println(StopGameServerSessionPlacementRequest.toJsonString(resp));
} catch (TencentCloudSDKException e) {
System.out.println(e.toString());
}
}
}
```

Note :

You can use API 3.0 Explorer for online debugging. You can select TencentCloud APIs under "Game Server Elastic-scaling" > "Service Management APIs" on the left sidebar, and perform operations such as "Code Generation" and "Online Call".

GSE Local

Last updated : 2021-06-28 10:09:36

GSE Local

GSE Local is a command line tool that can independently launch the game server hosting service GSE. This tool also provides runtime logs including the server initialization, health check, and API calls and responses.

GSE Local is limited to launch GSE hosting services and test your game integration on a local device, which will shorten the debugging time and improve efficiency at the iterative development of games. Otherwise, you have to upload each new game package to GSE and configure the server fleet to host games.

With GSE Local, you can test that:

- Your game server correctly integrates the GSE server development kit, properly communicates with GSE service, and is able to launch new game sessions, accept new players and report the running status.
- Your game client correctly integrates the GSE-related TencentCloud APIs to retrieve existing game sessions, launch new game sessions, and allow players to join and connect to the game session.

Setting Up GSE Local

GSE Local can run on Windows, Linux and Mac in any GSE-supported languages. You can download the installation package according to the operating system:

- GSE Local for Windows
- GSE Local for Linux
- GSE Local for Mac

> Note :
> The following sample code is applicable to Linux and MacOS. For Windows, we recommend to use the gitbash command line tool to run `curl` command.

Testing Game Server

If you only need to test your game server, directly use `curl` to simulate the game client calls to GSE Local and verify that your game server can complete the following operations as expected:

1. During launch, the game server will call the `ProcessReady` API to inform GSE that the server is ready to host a game server session.
2. During runtime, the game server will use the `onHealthCheck` callback to send its running status to GSE every minute.
3. The game server will respond to requests and trigger the `onStartGameServerSession` callback (call the `activateGameServerSession` API in this process) to launch a new game session.

Step 1: launch GSE Local

Open the command prompt window, navigate to the directory of `gselocal_windows`, `gselocal_linux` or `gselocal_mac`, and run the program. This document uses the Mac program `./gselocal_mac` as an example. After the program is launched, it will automatically connect to GSE Local.
Enter the following command in a terminal window:

```
./gselocal_mac
```

If the following information appears in the command prompt window, the launch is successful:

```
{"level":"info","ts":"2020-10-20T09:16:09.364+0800","msg":"start grpc v3 server success"}
```

Step 2: launch the game server

Launch the game process in a programming tool or command line tool. The game process then will call the `ProcessReady` API to prepare for hosting a session and print the following logs:

```
Getting process ready, LogPath: System.String[], ClientPort: 3237, GrpcPort: 6224
Process ready succeed, resp: { }
Server Start On Locolhost:6224
```

After receiving the ProcessReady request, GSE Local will also print logs and start the health check:

```
{"level":"info","ts":"2020-10-20T09:27:03.172+0800","msg":"ProcessReady Info is","pid":"41688","r
equestId":"3b38495b38bc4ef8a59ae8****a8256d","info":"clientPort:3237 grpcPort:6224 "}
{"level":"info","ts":"2020-10-20T09:27:03.172+0800","msg":"set runner success","pid":"41688","pro
cessUUID":"527bf89b-d128-4b5d-bfea-****3d22ede7"}
{"level":"info","ts":"2020-10-20T09:28:03.276+0800","msg":"onHealthCheck received","pid":"41688",
"health":true}
```

```
{"level":"info","ts":"2020-10-20T09:29:03.256+0800","msg":"onHealthCheck received","pid":"41688",
"health":true}
{"level":"info","ts":"2020-10-20T09:30:03.261+0800","msg":"onHealthCheck received","pid":"41688",
"health":true}
```

Step 3: use curl to create a game server session and a player session

Use `curl` to simulate the client calls. For specific parameters, see APIs.

Create a game server session

Run the following command to configure the `FleetId` parameter. You can set it to any valid strings `(^fleet-¥S+)` in GSE Local.

```
curl -d '{"Action":"CreateGameServerSession", "FleetId":"fleet-1235", "MaximumPlayerSessionCount"
:5}' http://127.0.0.1:8080/capi
```

The following log message displayed in the command prompt window indicates that GSE Local has sent the `onStartGameServerSession` callback to your game server. If a game server session is successfully created, your game server will call the `ActivateGameServerSession` API to respond to the callback. The logs are as follows:

```
{"level":"info","ts":"2020-10-20T09:37:08.580+0800","msg":"API to use: GSE.CreateGameServerSessio
n, with input","req":"FleetId:<value:¥"fleet-1235¥" > MaximumPlayerSessionCount:<value:5 > "}
{"level":"info","ts":"2020-10-20T09:37:08.580+0800","msg":"Reserved process: 41688 for GameServer
Session: qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-****6a09bff
e"}
{"level":"info","ts":"2020-10-20T09:37:08.580+0800","msg":"start to call StartGameSessionByGrpc t
o game server","gameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654
a-293b-4f1f-b71f-****6a09bffe"}
{"level":"info","ts":"2020-10-20T09:37:08.597+0800","msg":"onGameSessionActivate received","pid":
"4****","gameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-
4f1f-b71f-****6a09bffe","requestId":"de1a678dea364db4b487ff84ad****31"}
{"level":"info","ts":"2020-10-20T09:37:08.598+0800","msg":"call StartGameSessionByGrpc to game se
rver success","gameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a
-293b-4f1f-b71f-****6a09bffe"}
```

Querying a game server session

GSE Local uses `curl` to pass the game server session ID and object. Please note that the status of a new server session will change from “Activating” to “Active” after the game server calls the `ActivateGameServerSession` API. To view the status, run the following curl command to call the `DescribeGameServerSessions` API:

```
curl -d '{"Action":"DescribeGameServerSessions", "FleetId":"fleet-1235"}' http://127.0.0.1:8080/capi
```

The output is as shown below:


```
{"Response":{"GameServerSessions":[{"AvailabilityStatus":"Enable","CreationTime":"2020-10-20T01:37:08Z","CreatorId":"","CurrentCustomCount":0,"CurrentPlayerSessionCount":0,"DnsName":"","FleetId":"fleet-1235","GameProperties":[],"GameServerSessionData":"","GameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-2fa56a09bffe","InstanceType":"localhost","IpAddress":"127.0.0.1","MatchmakerData":"","MaxCustomCount":0,"MaximumPlayerSessionCount":5,"Name":"","PlayerSessionCreationPolicy":"ACCEPT_ALL","Port":3237,"Status":"ACTIVE","StatusReason":"","TerminationTime":null,"Weight":0}],"NextToken":"","RequestId":"s1603158295201357000"}}
```


Testing Game Server and Client

Prerequisites

You have completed the game server tests.

Step 1: add players

Run the following command to add players. The `GameServerSessionId` parameter is obtained in the response of the API used in creating a game server session

```
curl -d '{"Action":"JoinGameServerSession", "GameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-****6a09bffe", "PlayerId":"k****111"}' http://127.0.0.1:8080/capi
```

The GSE Local Command Prompt displays the following logs, indicating that the game server has sent the AcceptPlayerSession request to verify a new player connection.


```
{"level":"info","ts":"2020-10-20T10:03:43.096+0800","msg":"API to use: GSE.JoinGameServerSession, with input","req":"GameServerSessionId:¥"qcs::gse:local::gameserversession/fleet-****/gssess-c648654a-293b-4f1f-b71f-****6a09bffe¥" PlayerId:¥"ka****11¥" "}
{"level":"info","ts":"2020-10-20T10:03:43.096+0800","msg":"Creating player session with id: kadin111 for gameServersessionId: qcs::gse:local::gameserversession/fleet-****/gssess-c648654a-293b-4f1f-b71f-****6a09bffe"}
{"level":"info","ts":"2020-10-20T10:03:43.096+0800","msg":"Created player session with PlayerId: kadin111 and PlayerSessionId: psess-56dd6f48-08d4-4a11-9330-****09784977"}
```


Step 2: query a player session

Call the `DescribePlayerSessions` to query a player session. The initial status of the player session is “Reserved”:

- If the client successfully connects to the game server within 1 minute, the player session status will become “Active”.
- If the client fails to connect to the game server within 1 minute, the player session status will become “TIMEDOUT”.

```
curl -d '{"Action":"DescribePlayerSessions", "GameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gssess-c648654a-293b-4f1f-b71f-2fa56a09bffe", "PlayerId":"kadin111"}' http://127.0.0.1:8080/capi
```

The output is as shown below:

```
{"Response":{"NextToken":"","PlayerSessions":[{"CreationTime":"2020-10-20T02:03:43Z","DnsName":"","FleetId":"fleet-****","GameServerSessionId":"qcs::gse:local::gameserversession/fleet-1235/gsess-c648654a-293b-4f1f-b71f-****6a09bffe","IpAddress":"127.*.*.1","PlayerData":"","PlayerId":"ka****11","PlayerSessionId":"psess-56dd6f48-08d4-4a11-9330-****09784977","Port":3237,"Status":"TIMEDOUT","TerminationTime":"1970-01-01T00:00:00Z"}],"RequestId":"s16031596094****2000"}}%
```

Step 3: connect the client player to the server

After creating a game session and player session, you can directly use `localhost:port` to join a client player to the game session.

The GSE Local Command Prompt will display logs, indicating that the game server has sent the AcceptPlayerSession request to verify the new player connection. If you use `curl` to call the `DescribePlayerSessions` API, the player session status should be changed from “Reserved” to “Active”.

Step 4: send the test report to GSE

To verify that your game server sends the game and player statuses to GSE, ensure your game server always send these statuses to GSE Local to help GSE Local manage player needs and correctly report metrics. GSE Local will record the following actions. You may also need `curl` to track the status change.

- **A player disconnects from the game session**
  The GSE Local logs should display that the game server called the `RemovePlayerSession` API. The status in the response of the `DescribePlayerSessions()` API changed from “Active” to “Completed”. You can also call the `DescribeGameServerSessions` API to check that the current number of players in the game session has decreased by one.

- **The game session ends**
  The GSL Local logs should display that the game server called the `TerminateGameServerSession` API. The status in the response of the `DescribeGameServerSessions` API changed from “Active” to “Terminated” or “Terminating”.
- **The server process stops**
  The GSE Local logs should display that the game server called the `ProcessEnding` API.

Testing Game Client Calls to GSE

All game session and player session APIs used in game server tests and game server and client tests use `curl` to call GSE Local. You can use codes to call the following APIs in the game service to verify whether your game server is running properly. For the local debugging, you need to call `http://127.0.0.1:8080/capi` .

- CreateGameServerSession
- DescribeGameServerSessions
- JoinGameServerSession
- JoinGameServerSessionBatch
- DescribePlayerSessions

The GSE Local Command Prompt only displays the logs of the `CreateGameServerSession` API calls. As shown in the log message, GSE Local prompts the time when your game server launches a game session (using the `onStartGameServerSession` callback). After your game server uses the callback, GSE Local will obtain the ActivateGameServerSession response. You can `use` curl to view the calling of other APIs.

Notes

Take notice of the following points when using GSE Local:

1. Different from the GSE Web service, GSE Local does not track the running status or the `onProcessTerminate` callback triggering of the server. GSE Local only records the runtime report of the game server.
2. The FleetId will not be verified during the calling of Tencent Cloud development kid, because this parameter can be set to any valid strings `(^fleet-¥S+)` .
3. The game session created using GSE Local has a distinct ID structure, which contains `local` as shown below:

```
arn:gse:local::gamesession/fleet-****/gsess-56961f8e-db9c-4173-97e7-****82f0daa6
```

Latency Test Tool

Last updated : 2021-04-12 14:22:16

This document provides the addresses and examples for latency test in different regions. Both HTTPS and UDP addresses are supported.

HTTPS and UDP addresses for latency test in regions

Region	HTTPS Address	UDP Address
Beijing	https://ap-beijing.speed.tencentgse.com	ap-beijing.speed.tencentgse.com
Shanghai	https://ap-shanghai.speed.tencentgse.com	ap-shanghai.speed.tencentgse.com
Hong Kong (China)	https://ap-hongkong.speed.tencentgse.com	ap-hongkong.speed.tencentgse.com
Guangzhou	https://ap-guangzhou.speed.tencentgse.com	ap-guangzhou.speed.tencentgse.com
Chengdu	https://ap-chengdu.speed.tencentgse.com	ap-chengdu.speed.tencentgse.com
Singapore	https://ap-singapore.speed.tencentgse.com	ap-singapore.speed.tencentgse.com
Mumbai	https://ap-mumbai.speed.tencentgse.com	ap-mumbai.speed.tencentgse.com
Silicon Valley	https://na-siliconvalley.speed.tencentgse.com	na-siliconvalley.speed.tencentgse.com
Virginia	https://na-ashburn.speed.tencentgse.com	na-ashburn.speed.tencentgse.com
Frankfurt	https://eu-frankfurt.speed.tencentgse.com	eu-frankfurt.speed.tencentgse.com
Seoul	https://ap-seoul.speed.tencentgse.com	ap-seoul.speed.tencentgse.com
Tokyo	https://ap-tokyo.speed.tencentgse.com	ap-tokyo.speed.tencentgse.com

Example

Let’s take Guangzhou as an example.

- **HTTPS**

```
ping ap-guangzhou.speed.tencentgse.com
curl https://ap-guangzhou.speed.tencentgse.com/v1/ping
```

- **UDP**

```
Domain name + PORT (8888)
ap-guangzhou.speed.tencentgse.com + PORT (8888)
```

Game Process Launch Configuration

Last updated : 2021-04-12 14:22:16

Launching game process as a root user or user_00 in Linux environment

In Linux environment, the game process should be launched by a root user by default. If you want to launch the game process as a non-root user, please do the following:

1. Add the file `gse.yaml` to the root directory of the game's asset package, which means the decompressed file path will be `/local/game/gse.yaml` on the game server fleet instance;

2. The content of the file `gse.yaml` is shown below, indicating that user_00 is added to the `users` user group. You cannot configure other users and user groups currently;

   ```
   User: user_00:users
   ```

   When the file `gse.yaml` is added to the asset package, GSE will launch the game process with `user_00:users` and set the users and user groups of all files under `/local/game` as `user_00:users` .

See below for the example:

```
[root@VM-0-200-centos local]# tree -u -g game
game
├── [user_00  users   ]  gse_5dbe4ee9-e4ca-c7e1-6e8c-3bc856a3ba60.zip
├── [user_00  users   ]  gse.yaml
└── [user_00  users   ]  tarke
    └── [user_00  users   ]  linux
        ├── [user_00  users   ]  linux
        │   └── [user_00  users   ]  linux
        │       ├── [user_00  users   ]  log_2810.txt
        │       ├── [user_00  users   ]  log_2822.txt
        │       ├── [user_00  users   ]  log_2841.txt
        │       ├── [user_00  users   ]  log_2862.txt
        │       ├── [user_00  users   ]  log_2876.txt
        │       └── [user_00  users   ]  perfasset
        └── [user_00  users   ]  user00muli.zip

4 directories, 9 files
[root@VM-0-200-centos local]# ps -ef |grep perfasset
user_00     2810    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00     2822    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00     2841    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00     2862    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
user_00     2876    2754  0 19:43 ?        00:00:00 /local/game/tarke/linux/linux/linux/perfasset
root        4035    3220  0 19:49 pts/0    00:00:00 grep --color=auto perfasset
```

Executing `install.sh` before launching game process in Linux environment

Before a game process is launched, you may need to install some software or configure some environment variables on the CVM instance with the following steps:

1. Create the `install.sh` script and write the operations to be conducted before launching the game process in this script;
2. Add the file `install.sh` under the root directory of the game's asset package, which means the decompressed path will be `/local/game/install.sh` on the game server fleet instance.

Launch configuration for Java game process

In Linux environment, a command like `java -jar XXXX.jar` can be used to launch Java programs. The following configurations are required to ensure the Java game process is successfully launched:

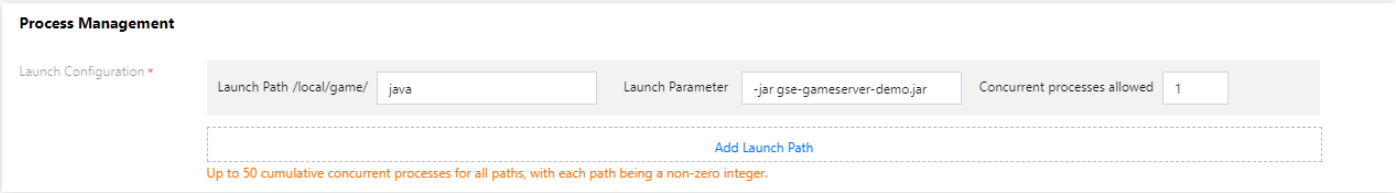
1. Write the `install.sh` script

```
#!/bin/bash
```

```
# Install the JDK 1.8 environment - y indicates answer yes for all questions
yum install java-1.8.0-openjdk.x86_64 -y
# Put the java command under `/local/game` with a soft link
ln -s /usr/bin/java /local/game/java
```

2. Put `install.sh` script under the root directory of the game's asset package, which means the decompressed path will be `/local/game/install.sh` on the game server fleet instance.

3. When creating the game server fleet, enter `/local/game/java` as the launch path, and enter `-jar jar package specified by user` as the launch parameter.



4. After the game process is successfully launched, the content of the path `/local/game` is shown below:

```
[root@VM-8-206-centos game]# ll
total 77032
-rw-r--r-- 1 root root 41270086 Jan 21 15:35 gse-gameserver-demo.jar
-rw-r--r-- 1 root root 37599574 Jan 21 23:31 gse_08063e02-f884-e503-e88c-c483673ac879.zip
-rwxr-xr-x 1 root root       70 Jan 21 15:35 install.sh
lrwxrwxrwx 1 root root       13 Jan 21 15:35 java -> /usr/bin/java
drwxr-xr-x 2 root root     4096 Jan 25 00:00 logs
```