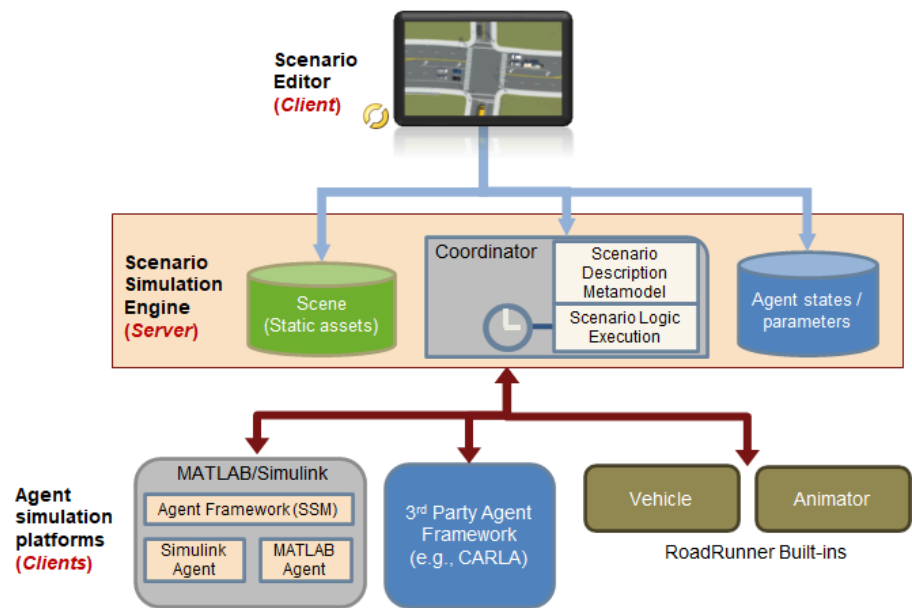


Scenario Simulation Engine (SSE)

- Overview of SSE co-simulation architecture and services
- Manage client-server connections
- Access static scene, map, and scenario logic models
- Configure and control co-simulation
 - SSE engine events
 - Client to server (gRPC) calls to configure a co-simulation
 - Client to server (gRPC) calls to manage a co-simulation
- Access runtime world states
- Log and debug co-simulation

Simulation of a scenario modelled as SSD is supported by the Scenario Simulation Engine (SSE). SSE provides a client-server co-simulation environment that allows multiple clients to connect and participate in multiple aspects of a scenario simulation. The following figure illustrates SSE and the client-server architecture at a high-level.



Overview of SSE co-simulation architecture and services

SSE is established on [Google gRPC](#), an universal framework capable of supporting


- Inter-process communication and network communication
- Multiple operating systems and language implementations (including C++, Python, Java, PHP etc.)
- Remote Procedure Call (RPC) and Publish/Subscribe communication semantics

With these features, clients of SSE can be implemented within different applications such as



- Within some modeling and simulation platforms: such as MATLAB/Simulink, RoadRunner, and CARLA
- A standalone application: such as an agent/actor behavior app created as a standalone executable
- A web app

In scenario v1 (22a), the RoadRunner app hosts both the SSE server and 3 in-app clients, including:

SSE Client in RoadRunner App	GUI
<ul style="list-style-type: none">A playback tool client that provide GUI to allow users to control the simulation	
<ul style="list-style-type: none">A viewer client that displays animation and simulation results	

SSE Client in RoadRunner App	GUI
<ul style="list-style-type: none">An actor controller client that provides built-in vehicle behaviors	 RoadRunner built-in vehicles

In addition, scenario v1 includes 2 external simulator clients, to support vehicle behaviors implemented in other platforms.

SSE Client outside RoadRunner App	GUI
<ul style="list-style-type: none">MATLAB/Simulink co-simulation client (available as a part of MATLAB/Simulink installation)	
<ul style="list-style-type: none">CARLA co-simulation client (available as a part of RoadRunner installation)	

SSE provides co-simulation services in the following 5 categories

1. Manage client-server communications
2. Manage and provide access to static contents including 3D scene, HD map, and the scenario logic model
3. Host co-simulation runs, manage playback control, agent simulators, and visualization clients to allow them contribute to a cohesive simulation
4. Maintain and provide access to runtime states and parameters of actors
5. Provide logging and debugging utilities

The following sections will focus on each of these categories. All the APIs are implemented as gRPC services. See [this page](#) for information on I/O arguments of these APIs.

Manage client-server connections

Client-server calls for connecting to SSE
<pre>//////////////////////////////////// // Client connection and subscriptions //////////////////////////////////// // Initialization function where calling clients are assigned an id. rpc RegisterClient (RegisterClientRequest) returns (RegisterClientResponse) {} // Registers the calling client as ready to proceed with the next update. rpc SetReady (SetReadyRequest) returns (SetReadyResponse) {} // Registers the calling client to receive events. // Note that this is a long-lived stream, and persists until the client or the server shuts down. rpc SubscribeEvents (SubscribeEventsRequest) returns (stream Event) {}</pre>

Access static scene, map, and scenario logic models

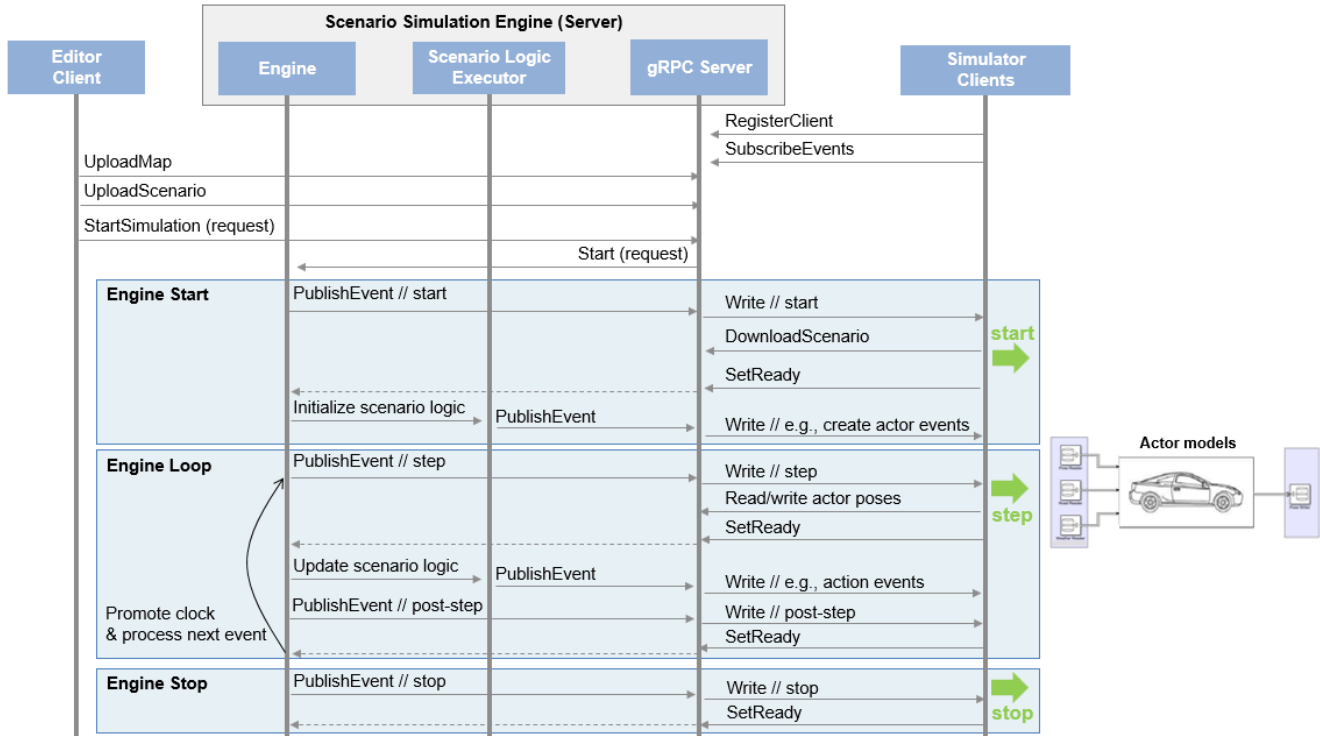
Client-server calls to upload/download scene, map, and scenario
<pre>//////////////////////////////////// // Scene and map access interface //////////////////////////////////// // Upload the HD map rpc UploadMap (UploadMapRequest) returns (UploadMapResponse) {} // Download the HD map rpc DownloadMap (DownloadMapRequest) returns (DownloadMapResponse) {} //////////////////////////////////// // Scenario access interface ////////////////////////////////////</pre>

```
// Uploads the provided scenario making it available for simulation.
// - Overwrites any previously uploaded scenario
rpc UploadScenario (UploadScenarioRequest) returns (UploadScenarioResponse) {}

// Download the most recently uploaded scenario
rpc DownloadScenario (DownloadScenarioRequest) returns (DownloadScenarioResponse) {}
```

Configure and control co-simulation

A co-simulation run orchestrated by SSE can be illustrated using the following sequence diagram.



SSE engine events

In above sequence diagram, communications from SSE (gRPC Server) to clients are implemented as engine events. The following are descriptions of these events.

Engine events from SSE (server) to clients

```
////////////////////////////////////
// Data model for simulation engine events
////////////////////////////////////

//
// Simulation engine events are sent by the Scenario Simulation Engine (SSE) to
// co-simulation clients. Some engine events request clients to perform certain
// operations so that co-simulation can happen in a synchronous and coherent fashion.
// These events require a co-simulation client to reply with a message that notifies
// the requested operation has completed.
//
message Event
{
    // Priority of this event, suggesting which event should be processed first
    // when multiple events occur at the same time.
    int32 priority = 1;

    // Whether this event requires a synchronous client to call 'SetReady'. After
    // publishing an event that requires 'SetReady', the simulation engine will wait
    // for all the synchronous clients to call 'SetReady', before it continues. This
    // ensures subsequent operations are based on a synchronous snapshot of the entire
    // scenario system.
    bool need_set_ready = 2;

    // Type specific attributes
```

```

oneof type {
    //////////////////////////////////////////
    // \name Events for managing client-server connections
    ///@{
    // Event to notify that gRPC co-simulation server will shutdown
    ServerShutdownEvent server_shutdown_event = 101;
    // Event to notify that an event subscriber has been added
    ClientSubscribedEvent client_subscribed_event = 102;
    // Event to notify that an event subscriber has been removed
    ClientUnsubscribedEvent client_unsubscribed_event = 103;
    ///@}

    //////////////////////////////////////////
    // \name Events for notifying changes in scene, scenario, simulation settings,
    // and simulation status
    ///@{
    // Event to notify that the scene (e.g., map) has changed
    SceneChangedEvent scene_changed_event = 111;
    // Event to notify that the scenario has changed
    ScenarioChangedEvent scenario_changed_event = 112;
    // Event to notify that the simulation settings have changed
    SimulationSettingsChangedEvent simulation_settings_changed_event = 113;
    // Event to notify that the simulation status has changed
    SimulationStatusChangedEvent simulation_status_changed_event = 114;
    ///@}

    //////////////////////////////////////////
    // \name Events for organizing a co-simulation with multiple clients
    ///@{
    // Event to notify that a simulation run is started. A synchronous co-simulation
    // client that listens to this event must reply with a SetReady call.
    SimulationStartEvent simulation_start_event = 121;
    // Event to notify that a simulation run will be ended. A synchronous co-
    // simulation client that listens to this event must reply with a SetReady call.
    SimulationStopEvent simulation_stop_event = 122;
    // Event to notify clients to promote simulation clock to the next step, and
    // perform necessary updates. A synchronous co-simulation client that listens to
    // this event must reply with a SetReady call.
    SimulationStepEvent simulation_step_event = 123;
    // Event to notify clients to optionally log simulation results of the current
    // step. A synchronous co-simulation client that listens to this event must reply
    // with a SetReady call.
    SimulationPostStepEvent simulation_post_step_event = 124;
    ///@}

    //////////////////////////////////////////
    // \name Events for notifying clients with scenario logic commands
    ///@{
    // Event to notify clients to create an actor. A synchronous co-simulation client
    // that listens to this event must reply with a SetReady call.
    CreateActorEvent create_actor_event = 201;
    // Event to notify clients that an actor will be destroyed. A synchronous co-
    // simulation client that listens to this event must reply with a SetReady call.
    DestroyActorEvent destroy_actor_event = 202;

    // Event to notify an actor to perform a specified action
    ActionEvent action_event = 211;
    ///@}
}
}

```

Client to server (gRPC) calls to configure a co-simulation

In the sequence diagram, client to server communications are implemented as gRPC service calls. Below are the service calls that configure a co-simulation. See also the [SimulationSettings](#) object for information on the parameters that you can configure.

Client to server calls to configure a co-simulation

```

////////////////////////////////////////
// Simulation settings access interface

```

```

////////////////////////////////////

// Set simulation settings
rpc SetSimulationSettings (SetSimulationSettingsRequest) returns (SetSimulationSettingsResponse) {}

// Request the server the change the pace of simulation
rpc SetSimulationPace (SetSimulationPaceRequest) returns (SetSimulationPaceResponse) {}

// Get simulation settings
rpc GetSimulationSettings (GetSimulationSettingsRequest) returns (GetSimulationSettingsResponse) {}

```

Client to server (gRPC) calls to manage a co-simulation

In the sequence diagram, clients request simulation to start/pause/step/stop by placing gRPC service calls to SSE. Below are descriptions of these call.

Client-server calls to manage a co-simulation

```

////////////////////////////////////
// Simulation control interface
////////////////////////////////////

// Requests that the simulation start.
// - Does nothing if the simulation is already running.
rpc StartSimulation (StartSimulationRequest) returns (StartSimulationResponse) {}

// Request to immediately stop current run and start off a new run.
rpc RestartSimulation (RestartSimulationRequest) returns (RestartSimulationResponse) {}

// A blocking call that requests for the simulation to stop. The RPC call returns with
// a response after simulation has stopped.
// - Does nothing if the simulation is not already running.
rpc StopSimulation (StopSimulationRequest) returns (StopSimulationResponse) {}

// A non-blocking call to request for the simulation to stop. The RPC call returns
// immediately with a response and does not wait until simulation has actually stopped.
// - Does nothing if the simulation is not already running.
rpc StopSimulationRequested (StopSimulationRequest) returns (StopSimulationResponse) {}

// If the simulation is paused, requests to advance the simulation by a single
// time step and remain paused.
// - This service record the single stepping request and return without waiting
//   for simulation to complete.
rpc StepSimulation (StepSimulationRequest) returns (StepSimulationResponse) {}

// Request that the simulation be paused or unpaused.
//
// Upon pause requested, simulation engine will:
// - Complete operations of the current time step, including the simulation step
//   event and the simulation post-step event
// - Advance time
// - Notify clients that simulation is paused (via a SimulationStatusChangedEvent)
// - Pause without proceeding
//
// Upon unpause requested, simulation engine will:
// - Notify clients that simulation is running (via a SimulationStatusChangedEvent)
// - Resume simulation
rpc ToggleSimulationPaused (ToggleSimulationPausedRequest) returns (ToggleSimulationPausedResponse) {}

// Get current simulation status
rpc GetSimulationStatus (GetSimulationStatusRequest) returns (GetSimulationStatusResponse) {}

// Get current simulation time
rpc GetSimulationTime (GetSimulationTimeRequest) returns (GetSimulationTimeResponse) {}

```

Access runtime world states

APIs to access runtime world states and scenario logic status

```

////////////////////////////////////

```

```

// World states access interface
////////////////////////////////////

// Submits actor runtime states to the server.
// Note that:
// - This call shall be placed as a part of event listener callback of
//   a simulation step event.
// - The set values will not be committed by the server until all the clients
//   has acknowledged the step event.
rpc SetRuntimeActors (SetRuntimeActorsRequest) returns (SetRuntimeActorsResponse) {}

// // Returns the requested actors' runtime attributes.
rpc GetRuntimeActors (GetRuntimeActorsRequest) returns (GetRuntimeActorsResponse) {}

// Returns the requested actors.
rpc GetActors (GetActorsRequest) returns (GetActorsResponse) {}

// Returns the requested phase's runtime status.
rpc GetPhaseStatus (GetPhaseStatusRequest) returns (GetPhaseStatusResponse) {}

```

Log and debug co-simulation

SSE logging and debugging APIs

```

////////////////////////////////////
// Simulation logging interface
////////////////////////////////////

// Query the server for communication log. The query can be to return all the messages, info, warning or errors,
// messages pertaining to a particular client etc.
rpc QueryCommunicationLog(QueryCommunicationLogRequest) returns (QueryCommunicationLogResponse) {}

rpc EnableCommunicationLogging(EnableCommunicationLoggingRequest) returns (EnableCommunicationLoggingResponse) {}

// Query the server for communication log. The query can be to return all the messages, info, warning or errors,
// messages pertaining to a particular client etc.
rpc QueryWorldRuntimeLog(QueryWorldRuntimeLogRequest) returns (QueryWorldRuntimeLogResponse) {}

rpc EnableWorldRuntimeLogging(EnableWorldRuntimeLoggingRequest) returns (EnableWorldRuntimeLoggingResponse) {}

// Add a diagnostic message to the server. The diagnostic message can be information, warning or error.
rpc AddDiagnosticMessage(AddDiagnosticMessageRequest) returns (AddDiagnosticMessageResponse) {}

// Query the server for diagnostic messages. The query can be to return all the messages, info, warning or errors,
// messages pertaining to a particular client etc.
rpc QueryDiagnosticMessageLog(QueryDiagnosticMessageLogRequest) returns (QueryDiagnosticMessageLogResponse) {}

```