Dartfish Live Collaboration Service API Description

Document revision 2.0 - Monday, 9 March 2020

1. INTRODUCTION

The role of the Dartfish Live Collaboration (DLC) service is essentially to expose a set of centralised annotated timelines 1. from which connected clients can retrieve information, 2. by which connected clients can be notified of changes and 3. to which connected clients can contribute by adding or editing information.

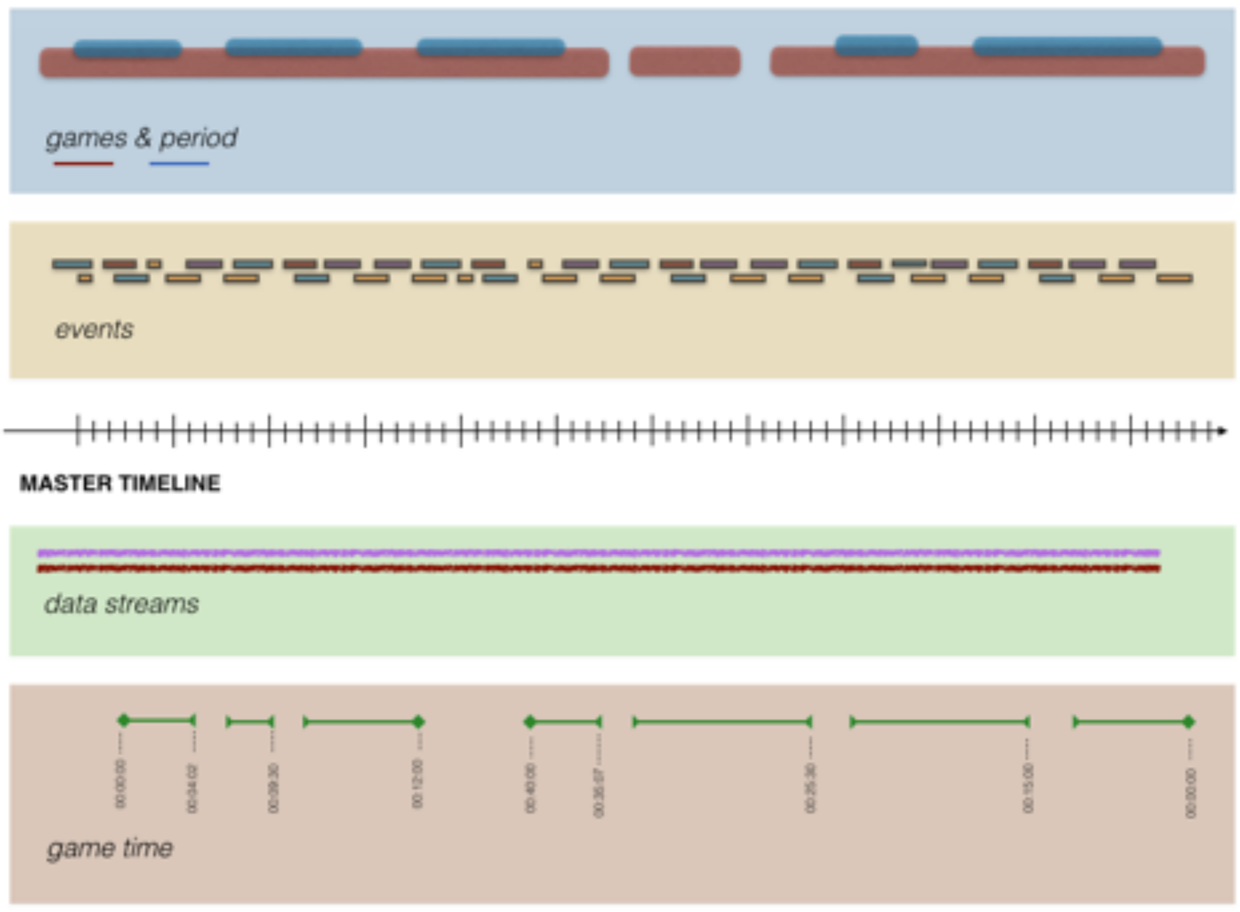

Figure 1: the four entities supported by the Dartfish Live Collaboration service

As depicted in Fig.1, the DLC timelines supports four type of entities:

- ***Events*** - Events are segments of time to which metadata such as label and keywords can be attached. Events are used to define and annotate phases during a competition.
- ***Games and Periods*** - Games and periods are used to structure a competition at a macroscopic level. Games are as non-overlapping segments of times that can be further split down in non-overlapping periods. Both games and periods can be annotated with metadata.
- ***Game Time*** - A game time describes the official time pace of a competition. It is defined by a series of start, pause, resume and ends with a stop point.
- ***Data streams*** - A data stream is a flow of data.

Note that each entity is independent from any other. There is no hierarchy link or rule enforced between or across them. For instance a game time is not limited to the bounds of a game or the insertion of an event does not require that a period or a game is running.

A Live Collaboration service supports more than one timeline at a time. Each individual timeline is embedded in a so-called *tagging session*. Supporting several independent timelines/sessions in parallel is useful in situations where several competitions happen at the same time in a same arena (such as for parallel judo matches where each mat is assigned its own timeline) . Each timeline/ session is completely isolated from the others.

Dartfish provides two different APIs to allow third-party clients to connect and interact with a LiveCollaboration service namely a .NET library and a gRPC protobuf service description.

The first API is provided through a .NET Standard nuget package exposing C# classes. It provides a higher level of abstraction than the gRPC-based API and is recommended for integration with programming languages supported by .NET and targeting platforms supporting .NET Core.

The second API is gRPC-based and is provided through a Google language agnostic *protobuf* file that describes the DLC service interface and its data. Given the *protobuf* file, a client proxy can be generated in a large range of programming languages (C++, Java, Python, Objective-C, C#,..).

2. API 1: C# .NET STANDARD LIBRARY

OVERVIEW

The API presented in this section is provided under the form of a C# .NET Standard-compatible nuget package. The package exposes C# helper classes that are essentially wrapper classes around the gRPC-based client proxy that constitutes the second API described in this document. These helper classes provide a higher level of abstraction, hides a great deal of complexity and isolates from gRPC implementation details. In particular,

- They provide an automatic clock correlation mechanism between the master clock of the DLC service and the local clock of the client. This allows the client to use his local time reference when communicating timing information to the DLC service through the helper classes. The helper class will translate the local time into master time.
- They hide gRPC-generated data classes and expose easily integrable C# data classes.
- They hide the tedious gRPC-based notification mechanism (long-lived pseudo-polymorph stream) and expose C# events to which client can easily subscribe individually.
- They provide an automatic reconnection mechanism to the DLC service in case of network disruption.

SESSION MANAGEMENT

Connecting to a DLC service, managing sessions and joining a particular session is performed through the class *TaggingSessionBroker*. The main methods exposed by *ITaggingSessionBroker* are described hereafter.

```
Task ConnectAsync(string endPoint, CancellationToken cancellationToken);
```

Connect to a DLC service specified by its endpoint. The endpoint format is I*Pv4 address:Port number.*

`Task DisconnectAsync();`

Disconnect from the DLC service.

`Task<IEnumerable<TaggingSessionInformation>>`
`GetAvailableTaggingSessionsAsync(CancellationToken cancellationToken);`

Returns the list of sessions currently available.

`Task<string> CreateTaggingSessionAsync(TaggingSessionInformation sessionInformation, CancellationToken cancellationToken);`

Create a new session with a description given by *sessionInformation.* The method returns an id uniquely identifying the session under the form of a string, The caller can suggest an id in the *sessionInformation* argument but if it is already assigned the service will create a random unique id.

`Task CloseTaggingSessionAsync(string sessionId, CancellationToken cancellationToken);`

Close the session specified by *sessionId.* Upon successful completion, the session is not advertised anymore, cannot be joined anymore and will be destroyed when all the clients of the session have left the session.

`event SessionListChangedHandler SessionListChanged;`

When registered, notifies when the list of available sessions has changed.

`Task<ITaggingSessionClient> JoinSessionAsync(string sessionId, string clientName, CancellationToken cancellationToken);`

Join the session specified by its *sessionId*.

Returns a *ITaggingSessionclient* that be used to interact with the specified session and associated timeline as described below.

INTERACTING WITH A SESSION/TIMELINE

Given a *ITaggingSessionClient*, a client can start interacting with the timeline owned by the session that he has joined.

1. SESSION STATE

`Task<List<SessionClient>> GetSessionClientsAsync(CancellationToken cancellationToken);`

Retrieve the list of the clients currently connected to the session

`event SessionClientsChangedHandler SessionClientsChanged;`

When registered, notifies when the list of the clients connected to the session changes.

`event SessionClosedHandler SessionClosed;`

When registered, notifies when the session closes.

`Task LeaveSessionAsync(CancellationToken cancellationToken);`

Leave the session and unsubscribes. From that point on, the instance of *ITagginSessionClient* is not usable anymore.

`ConnectionState ConnectionState { get; }`

Retrieve the current connection state with the service.

`event ConnectionStateChangedHandler ConnectionStateChanged;`

When registered, notifies when the state of the connection with the session changes.

2. EVENTS

Events are segments of time which can be annotated with metadata such as label and keywords. Events are used to define and annotate phases during a competition.

The functionality of game and period is carried through the *ITaggingEventHandler* interface of the *ITaggingSessionClient*.

‣ Editing

```
Task<string> AddTaggingEventAsync(EventDescription eventInformation, CancellationToken
cancellationToken);
```

Create a tagging event with timing information and metadata described in *EventDescription*.

Returns an id which uniquely identifies the event.

```
Task RemoveTaggingEventAsync(string eventId, CancellationToken cancellationToken);
```

Remove a tagging event identified by its id from the timeline.

```
Task UpdateTaggingEventAsync(string eventId, EventDescription eventInformation,
CancellationToken cancellationToken);
```

Update the timing or metadata of a tagging event identified by its id.

‣ Notifications

```
event TaggingEventAddedHandler TaggingEventAdded;
event TaggingEventUpdatedHandler TaggingEventUpdated;
event TaggingEventRemovedHandler TaggingEventRemoved;
```

When registered, notify when a tagging event has been added, removed or modified

‣ Retrieval

```
Task<EventDescription> GetTaggingEventAsync(string eventId, CancellationToken
cancellationToken = default(CancellationToken));
```

Retrieve the timing information and metadata of a tagging event identified by its id.

```
Task<IDictionary<string, EventDescription>> GetTaggingEventsAsync(long rangeIn, long
rangeOut, CancellationToken cancellationToken);
```

Retrieve tagging events whose start time point is comprised in the specified time range [*rangeIn, rangeOut*].

‣ Event Description

The class that describes an event is *EventDescription*:

```
  public class EventDescription
  {
    public string Label { get; set; }
    public string Description { get; set; }
    public string Color { set; get; }
    public long CueIn { set; get; }
    public long CueOut { get; set; }
    public List<List<string>> Keywords { get; set; }
  }
```

The *Color* field is a string that can only takes value from “Color1" to “Color8”. The *CueIn* and CueOut *fields* defines the time range of the event in local time reference. The *Keywords* field, in its simplest embodiment, is a list of pairs of strings where the first string of each pair is the category while the second string of the pair is the value. An example of *Keywords* could be {{“Player”, “John Doe”}, {“Phase, “Offensive”}} to specify that the event relates to “John Doe” for the category “Player” and to “Offensive” for the category “Phase” i.e. the player John Doe has the ball during an offensive phase.

3. GAMES & PERIODS

Games and periods are used to structure a timeline at a macroscopic level. Games are non overlapping segments of time that can optionally be further split down into non-overlapping periods (referenced henceforth as the no overlap rule). If any, a period must be fully contained in a parent game (referenced henceforth as the containment rule). A game can be period-less but a period cannot be game-less. Both games and periods can be annotated with metadata.

The functionality of game and period is carried through the *IGamePeriodHandler* interface of the *ITaggingSessionClient*.

‣ Editing

`Task<string> InitiateGameAsync(long atTime = -1, CancellationToken cancellationToken);`

Initiate a game at the specified time *atTime*. If not specified or -1, the game will start at the current time.

Returns an id that uniquely identifies the game.

Throws an exception if a game is already running or if the specified *atTime* violates the no overlap rule.

`Task EndGameAsync(long atTime = -1, CancellationToken cancellationToken);`

End a game at the specified time *atTime*. If not specified or -1, the game will end at the current time.

Throws and exception if no game is currently running or if the specified *atTime* is smaller than the start time of the game.

`Task AnnotateGameAsync(string gameId, Metadata metadata, CancellationToken cancellationToken);`

Annotate a game given its id *gameId*. The method overwrites the metadata previously associated with the game with the *metadata* argument. If only an incremental update of the metadata is desired, it is the client’s responsibility to retrieve the game metadata with the method *GetGameAsync*, to update it locally as desired and then call the method.

Throws an exception if *gameId* is invalid.

`Task<string> InitiatePeriodAsync(long atTime = -1, CancellationToken cancellationToken);`

Initiate a period at the specified time *atTime*. If not specified or -1, the game will start at the current time.

Returns an id that uniquely identifies the period and the parent game it belongs to.

Throws an exception if no game is currently running, if a period is already running or if the specified *atTime* violates the no overlap or containment rule.

```
Task EndPeriodAsync(long atTime = -1, CancellationToken cancellationToken);
```

End a period at the specified time *atTime*. If not specified or -1, the period will end at the current time.

Throws and exception if no period is currently running or if the specified *atTime* is smaller than the start time of the period

```
Task AnnotatePeriodAsync(string periodId, Metadata metadata, CancellationToken
cancellationToken);
```

Annotate a period given its id *periodId*. The method overwrites the metadata previously associated with the period with the content of the *metadata* argument. If only an incremental update of the metadata is desired, it is the client's responsibility to retrieve the period metadata with the method *GetPeriodAsync*, to update it as desired and then call the method.

Notifications

```
event GameInitiatedHandler GameInitiated;
event GameEndedHandler GameEnded;
event GameAnnotatedHandler GameAnnotated;
```

When registered, notify when a game has been initiated, ended or annotated.

```
event PeriodStartedHandler PeriodInitiated;
event PeriodEndedHandler PeriodEnded;
event PeriodAnnotatedHandler PeriodAnnotated;
```

Retrieval

```
Task<string> GetRunningGameIdAsync(CancellationToken cancellationToken);
```

Retrieve the id that uniquely identifies the current running game. The id can be used to annotate the game or retrieve a game description.

Returns the id of the running game or null if no game is running.

```
Task<IGame> GetGameAsync(string gameId, CancellationToken cancellationToken);
```

Retrieve a game identified by its *gameId*.

Throws an exception if the specified *gameId* is invalid.

```
Task<IPeriodId> GetRunningPeriodIdAsync(CancellationToken cancellationToken);
```

Retrieve the id that uniquely identifies the current running period. The id can be used to annotate the period or retrieve a period description.

Returns the id of the running period or null if no period is running.

```
Task<IGame> GetPeriodAsync(IPeriodId periodId, CancellationToken cancellationToken);
```

Retrieve a period identified by its *periodId*.

Throws an exception if the specified *periodId* is invalid.

4. GAME TIME

A game time describes the official time pace of a competition with respect to the centralised time. It is defined by a series of start, pause, resume and ends with a stop point. Some rules are enforced:

- Game times cannot overlap i.e. there is only one game time at a time (referred henceforth as the no overlap rule)
- It is expected that a game time is defined by a sequence of the following commands at progressing times: *Start/Pause/Resume/Pause/Resume/../Finish* (referred henceforth as the progression rule).

The functionality of game time is carried through the *IGameTimeHandler* interface of the *ITaggingSessionClient*.

‣ Editing

`Task<string> StartGameTimeAsync(GameTimeType type, long gameTimeFrequency, long time, long gameTime, CancellationToken cancellationToken));`

Start a game time . The argument *type* species if the game time is CountUp or CountDown. The *gameTimeFrequency* argument defines the game clock frequency (ticks/sec). The *time* argument specifies the position on the local timeline while the *gameTime* argument defines the initial value of the game time.

Returns an id that uniquely identifies the game time.

Throws an exception if a game time is already running or if the specified *time* argument violates the no overlap rule.

`Task PauseGameTimeAsync(long time, long gameTime, CancellationToken cancellationToken);`

Pause the current game time. The *time* argument defines the position on the local timeline while the *gameTime* argument specifies the game time at which the game clock is paused.

Throws an exception if no game time is currently running, if the running game time is already paused or if the specified *time* violates the progression rule.

`Task ResumeGameTimeAsync(long time, CancellationToken cancellationToken);`

Resume the current game time. The *time* argument defines the position on the local timeline. The game time resumes automatically from its previous pause value.

Throws an exception if no game time is currently running, if the running game time is already resumed or if the specified *time* violates the progression rule.

`Task StopGameTimeAsync(long time, long gameTime, CancellationToken cancellationToken);`

Stop the current game time. The *time* argument defines the position on the local timeline. The *gameTime* argument specifies the final game time. Once stopped, a game time cannot be edited anymore.

Throws an exception if no game time is currently running or if the specified *time* violates the progression rule.

`Task CorrectGameTimeAsync(long deltaTime, CancellationToken cancellationToken);`

Correct the time of the last command of the current game time. Positive *deltaTime* values correct for early trigger while negative values correct for late trigger. The *deltaTime* is expressed in local reference time. [The corresponding game time is also corrected if the corrected command corresponds to a Pause].

‣ Notifications

```
event GameTimeTransitionedHandler GameTimeTransitioned;
event GameTimeCorrectedHandler GameTimeCorrected;
```

When registered, notify when a game time changes state or when a game time is corrected.

‣ Retrieval

```
Task<string> GetCurrentGameTimeIdAsync(CancellationToken cancellationToken);
```

Retrieve the id of the current game time if any. The id can be used to retrieve the game time.

Returns the current game time id or null if no current game time.

```
Task<IGameTime> GetGameTimeAync(string gameTimeId, CancellationToken);
```

Retrieve a game time given its *gameTimeId*.

Throws an exception if the specified *gameTimeId* is invalid.

‣ Game time structure

```
  public interface IGameTime
  {
    string Id { get; }
    GameTimeType Type { get; }
    List<IGameTimePoint> GameTimePoints { get; }
    long Frequency { get; }
    GameTimeState State { get; }
  }

  public interface IGameTimePoint
  {
    long Time { get; }
    long GameTime { get;}
  }
```

A game time is represented as a series of game time points. Each time point is a pair of values, the first is the reference local time while the second is the game time. The list of game time points contains only time points corresponding to a start/pause/resume/stop transition. The first time point corresponds to the start, the last to the stop. Intermediate points correspond to alternating pause and resume transition. From a list of game time points, it is possible to determine the game time for each value of the reference local time by interpolation.

5. DATA STREAMS

Data streams are not yet supported by the DLC service.

CLOCK

The timeline exposed by the service relies on a centralised time that itself relies on a master clock maintained by the service itself. All time fields in client’s request to the service should be expressed in master time and all time fields in service responses or notifications are expressed in master time.

The class *TaggingSessionClient* is hiding this complexity and automatically translates local times into master time before calling the service and master times into local times from the service responses and notifications.

To allow this master-to-local and local-to-master time translation, *TaggingSessionClient* must be given the local clock which acts as the reference for the local time:

`IClock Clock { set; }`

Assigns a local clock so as to enable the instance to estimate the correlation between the local time and the master time.

```
public interface IClock
{
  long Time { get; }
  long Scale { get; }
}
```

The *IClock* interface has to be implemented by the client and its instance assigned to *TaggingSessionClient*. The *Time* field must be a linearly strictly increasing number of ‘ticks’ over time. It should return 0 to indicate that the clock has not started yet. The *Scale* field represents the expected number of ‘ticks’ per second and should be an invariant.
For instance, a client that is laying out events along a video stream whose timestamps are expressed in 90 KHz ticks will provide a clock whose 0 corresponds to the start of the stream and that increases by 90’000 ticks per seconds (i.e. *Scale* is 90’000). As another example, a client that is using NTP timestamps to layout events will provide a *IClock* instance locked to a NTP synchronised clock.

Note that a game clock cannot be used as *IClock* implementation if the game clock is allowed to pause and resume since the condition of strictly increasing clock is not respected.

PSEUDO-CODE EXAMPLE

The following example demonstrates how to connect to a DLC service, choose a session, connect a local clock and add an event to the session’s timeline.

```
// instantiate and connect a TaggingSessionBroker
string serverAddress = "192.168.1.128"; // change with the address of the server
System.Net.IPEndPoint serverEndPoint = new
System.Net.IPEndPoint(System.Net.IPAddress.Parse(serverAddress), 50051);
Dartfish.Component.TaggingSession.TaggingSessionBroker broker = new
Dartfish.Component.TaggingSession.TaggingSessionBroker();
await broker.ConnectAsync(serverEndPoint);

// retrieve available session list
var taggingSessions = await broker.GetAvailableTaggingSessionsAsync();

// just get first available session
string sessionId = taggingSessions.First()?.Id;

// join session and retrieve a client interface
Dartfish.Component.TaggingSession.ITaggingSessionClient client = await
broker.JoinSessionAsync(sessionId, "SpeedSkatingApp");
```

```
// assign the local Dartfish.Component.TaggingSession.IClock
// so that clock synchronisation can be performed
var clock = new LocalClock();
client.Clock = clock;

// push an event to the server
Dartfish.Component.TaggingSession.EventDescription eventInformation = new
Dartfish.Component.TaggingSession.EventDescription
{
  Label = "myEvent",
  Description = "myEvent Description",
  Color = "Color6",
  CueIn = clock.Time,                   // example: insert at current time
  CueOut = clock.Time + clock.Scale,    // 1 sec. duration
  Keywords = new System.Collections.Generic.List<string> { "speed", "45.34",
"acceleration", "3.67" }
};


await client.AddTaggingEventAsync(eventInformation);
```

3. API 2 : GRPC API

OVERVIEW

The gRPC api is a low-level api since it communicates directly with the service. The developer using this api will need to deal with the nuts and bolts of the gRPC service. In particular, he will need to implement the clock correlation mechanism and, if notification is necessary, handle directly the long-lived notification stream.

SESSION MANAGEMENT

`rpc GetAvailableTaggingSessions(rpcEmpty) returns (rpcTaggingSessions) {}`

Return a list of the currently available sessions.

`rpc CreateTaggingSession(rpcSessionInformation) returns (rpcSessionId) {}`

Create a new session based on the information passed in argument. It returns the newly created session Id.

`rpc CloseTaggingSession(rpcSessionId) returns (rpcEmpty) {}`

Close a tagging session identified by its id. Once closed, the session is not advertised anymore and cannot anymore be subscribed to. Session remains alive until no more client is connected to the session

In order to receive a notification when a change in the available session list occurs, the client need to subscribe to the service:

`rpc SubscribeToService(rpcSubscriptionToken) returns (stream rpcServiceNotification) {}`

Subscribe a client with subscription token to service-level notification. The subscription token is merely a unique token created by the client to be able to unsubscribe later. The method returns a stream that is maintained alive/open by the service and to which the service push notifications when change occurs at the service level.

`rpc UnsubscribeFromService(rpcSubscriptionToken) returns (rpcEmpty) {}`

Unsubscribe client with given subscription token from receiving service-level notifications

`rpc JoinSession(rpcRegistrationRequest) returns (rpcRegistrationToken) {}`
Join a particular session. Returns a registration token that allows to interact with the session.

`rpc LeaveSession(rpcRegistrationToken) returns (rpcEmpty) {}`
Leave a previously joined session.

INTERACTING WITH A SESSION/TIMELINE

1. SESSION STATE

`rpc GetSessionClients(rpcRegistrationToken) returns (rpcSessionClients) {}`
Return a descriptive list of the clients currently connected to the session.

`rpc SubscribeToSession(rpcRegistrationToken) returns (stream rpcSessionNotification) {}`
Subscribe to session-level notifications.The method returns a stream that is maintained alive/open by the service and to which the service pushes notifications when change occurs at the session level.

`rpc UnsubscribeFromSession(rpcRegistrationToken) returns (rpcEmpty) {}`
Unsubscribe from session-level notifications. The notification stream is closed by the service.

2. EVENTS

‣ Editing

`rpc AddTaggingEvent(rpcAddEventRequest) returns (rpcTaggingEventId) {}`
Add an event on then timeline as described by *rpcAddEventRequest.*.

Returns the id of the event assigned by the service for further reference.

`rpc RemoveTaggingEvent(rpcIdentifyEventRequest) returns (rpcEmpty) {}`
Remove an event identified by its id.

`rpc UpdateTaggingEvent(rpcUpdateEventRequest) returns (rpcEmpty) {}`
Update an event identified by its id with information contained in *rpcUpdateEventRequest*.

‣ Retrieval

`rpc GetTaggingEvent(rpcIdentifyEventRequest) returns (rpcTaggingEvent) {}`
Retrieves an event specified by its id. Returns a description of the event.
`rpc GetTaggingEvents(rpcGetEventsRequest) returns (stream rpcTaggingEvent) {}`
Retrieves tagging events whose start point is comprised between a specified time range. Returns a stream of event descriptions.

3. GAMES & PERIODS

‣ Editing

```
rpc InitiateGame(rpcInitiateGameRequest) returns (rpcGameId) {}

rpc EndGame(rpcEndGameRequest) returns (rpcEmpty) {}

rpc AnnotateGame(rpcAnnotateGameRequest) returns (rpcEmpty);

rpc InitiatePeriod(rpcInitiatePeriodRequest) returns (rpcPeriodId) {}

rpc EndPeriod(rpcEndPeriodRequest) returns (rpcEmpty) {}

rpc AnnotatePeriod(rpcAnnotatePeriodRequest) returns (rpcEmpty);
```

‣ Retrieval

```
rpc GetRunningGameId(rpcRegistrationToken) returns (rpcGameId) {}

rpc GetGame(rpcGetGameRequest) returns (rpcGame);

rpc GetRunningPeriodId(rpcRegistrationToken) returns (rpcPeriodId) {}

rpc GetPeriod(rpcGetPeriodRequest) returns (rpcPeriod);
```

4. GAME TIME

‣ Editing

```
rpc StartGameTime(rpcStartGameTimeRequest) returns (rpcGameTimeId);

rpc PauseGameTime(rpcPauseGameTimeRequest) returns (rpcEmpty);

rpc ResumeGameTime(rpcResumeGameTimeRequest) returns (rpcEmpty);

rpc StopGameTime(rpcStopGameTimeRequest) returns (rpcEmpty);

rpc CorrectGameTime(rpcCorrectGameTimeRequest) returns (rpcEmpty);
```

‣ Retrieval

```
rpc GetCurrentGameTimeId(rpcRegistrationToken) returns (rpcGameTimeId);

rpc GetGameTime(rpcGetGameTimeRequest) returns (rpcGameTime);
```

5. DATA STREAMS

Data streams are not yet supported by the Live Collaboration service

CLOCK SYNCHRONISATION

The timeline exposed by the service relies on a centralised time that itself relies on a master clock maintained by the service itself. All time fields in client's request should be expressed in master time and all time fields in service responses or notifications are expressed in master time. No assumption can be made on the master clock except that it is a linear strictly increasing function over time.
In the remainder of this document the terms master time and centralised time shall be used interchangeably.

The current master time can be accessed by calling:

```
rpc GetSessionTime(rpcRegistrationToken) returns (rpcSessionTime)
```

The response contains the current master time value and the scale/unit of the time. The *scale* field gives the frequency of the master clock i.e. the amount *time* is expected to increase in one second. The scale is invariant during the lifetime of the session.

```
message rpcSessionTime
{
  int64 time = 1;
   int64 scale = 2;
}
```

Each client is responsible for characterising the relation between its local clock and the master clock. Since it is assumed that the local clock is also a linear strictly increasing function over time and that the scales of both the master clock and the local clock are known, it is therefore enough to characterise the offset between the two clocks in order to be able to convert a local time in master time and vice versa.

Once the offset `clockOffset` between the two clocks is known, the conversion from local time to master time is

```
masterTime = (masterClockScale / localClockScale) * localTime + clockOffset
```

and the conversion from master time to local time is:

```
localTime = (masterTime - clockOffset) * localClockScale / masterClockScale)
```

The offset between the two clocks can be estimated by 1. collecting a series of master times and corresponding local times and then 2. running a statistical estimation based on the collected samples.

To perform the first operation, gather 20 time pair samples in the following way:

```
long[20][2] timeSamples;
int sampleIndex = 0;

for (i = 0;  i<20; i++)
{
  long localTimeTx = localClock.Time;
  long masterTime = GetSessionTime();
  long localTimeRx = localClock.Time;
  long localTimeAv = (localTimeRx + localTimeRx) / 2

  timeSamples[i][0] = localTimeAv;
  timeSamples[i][1] = masterTime;
}
```

Use simple linear regression on the time samples so as to estimate the offset:

```
long accMasterTime = 0, accLocalTime = 0;
 for (i=0; i< 20; i++)
{
  accLocalTime += timeSamples[i][0];
  accMasterTime += timeSamples[i][1]
}

clockOffset = (accMasterTime – (accLocalTime / localClockScale) * 
masterClockScale) / 20;
```

NOTE ABOUT NOTIFICATION STREAMS

Upon reception of a notification stream, the client must listen to the stream for notification. Implementation can vary depending the programming language.

```
while (await notificationStream.MoveNext())
{
  rpcSessionNotification notification = notificationStream.Current;
  HandleSessionNotification(notification);
}
```

The instruction **`await notificationStream.MoveNext()`** blocks until a notification is pushed by the service. The client then handles the notification and resumes listening. Upon unsubscribing, the stream closes and MoveNext returns false. Upon network failure or service shutdown, **`notificationStream.MoveNext()`** throws an exception that must be handled gracefully.