Introduction to the Game Synchronization SE

The FIcontent Specific Enablers are owned by different partners. Therefore, different Terms and Conditions might apply. Please check for each FIcontent Specific Enabler the Terms and Conditions [http://mediafi.org/?portfolio=game-synchronization-se#tab-terms-conditions] attached.

Game Synchronization SE Core

- *type of this SE: C# library*
- *Synchronizes the game content among players*
- *Provides RTS Lockstep game networking model*

Intended Audience

This specification is intended for application developers. The document provides a specification of how to interoperate with the Game Synchronization SE API. To use this information, the reader should firstly have a general understanding of Network programming. You should also be familiar with:

- Unity Networking
- Deterministic Simulation

Provider of the Game Synchronization SE

- Marcel Lancelle, ETHZ
- Chino Noris, DRZ
- Mattia Ryffel, DRZ

API Change History

Revision Date	Changes Summary
Sep 13, 2013	Version APIv0.1 drafted
Apr 01, 2015	Version APIv1.0 updated

How to Read This Document

Some special notations are applied to differentiate some special words or concepts. The following list summarizes these special notations.

- An italic font is used to represent document titles or some other kind of special text, e.g., URI.
- A bold font is used to represent class names and interfaces.
- A bold and italic font is used to represent methods, properties and attributes.

Additional Resources

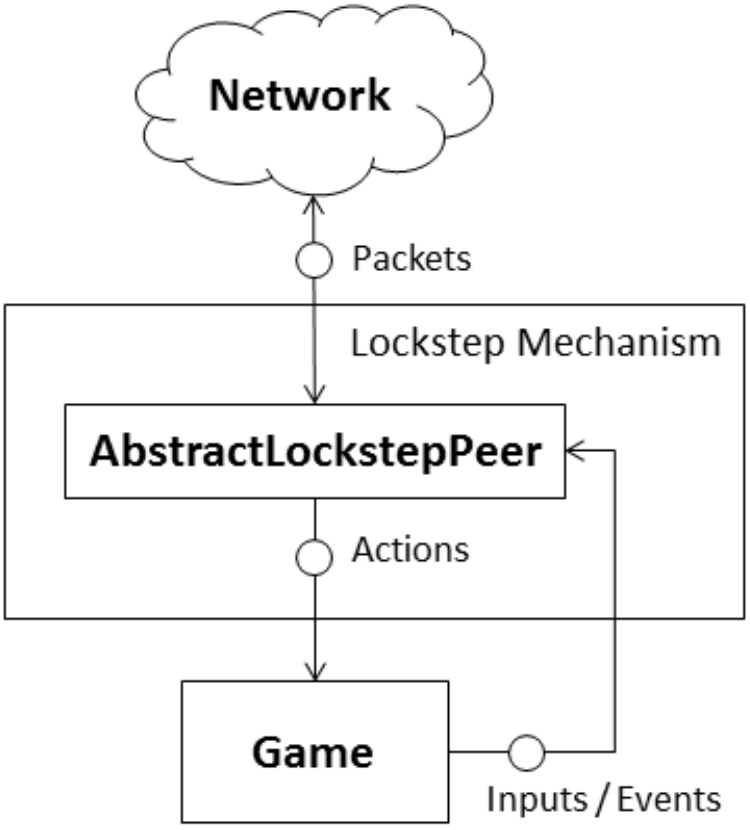
You can download the most current version of this document from the FIcontent platform documentation website at the Roadmap of FIcontent Gaming Specific Enablers. For more details about the usage of the Game Synchronization SE within the FIcontent platforms, please refer to the FIcontent Architecture, in specific the architecture of the Pervasive Games Platform.

For more details about getting started with the Game Synchronization SE within your own application or service, please refer to the Developer's Guide as a first starting point.

Overview of the Game Synchronization SE

The main functionality that the Game Synchronization SE provides is:

- RTS Lockstep mechanism for peer-to-peer synchronization of player's actions throughout the game.



Game Synchronization SE API Specification

RTS-LockStep Model

Disclaimer: A crucial requirement of the RTS-Lockstep model is that your game simulation must be deterministic, i.e. the same inputs should always produce the same output on all devices you plan to support. If this requirement is not met, the simulations will diverge, leading to different states of the game on different clients. Recovering from such de-synchronization is often complicated. If your game cannot be simulated deterministically, consider changing to a different networking model. Please note that particular care must be taken if you are using floating point computations, as the same computation can lead to different results depending on the specific model of processor you have.

In the RTS-Lockstep model, the concept is that the clients participating to a game session only communicate the actions of the players, and then locally simulate the effect of those. Each peer participating to the game is identified by a unique integer called *GUID*. The data sent between peers consist of actions associated with a *snap*, an integer counting the

number of simulation steps from the start of the game. When a player gives an input on a client, the client registers this action and schedules it to be executed in the near future. All clients then share the actions that are about to happen, and when a client has all the actions of all players associated with a particular snap, it executes a simulation step.

The following sections include more details about this mechanism and highlight the main actors.

Reliable delivery of packets

To simplify the integration and portability of this SE, we implemented it using the Unity Networking as the base transportation layer. The messages are sent via RPCs (remote procedure call), which have a reliable implementation in Unity.

Lockstep implementation overview

```
public partial abstract class AbstractLockstepPeer : MonoBehaviour
```

The main class of the Lockstep mechanism is provided by **AbstractLockstepPeer** class. This class provides the methods to connect and reliably send data, and implements the lockstep mechanism. When using the Game Synchronization SE, this abstract class must be implemented and the following methods must be overridden for your game specific purposes: **ExecuteAction** and **OnSimulationStarted**.

Lockstep mechanism

```
public uint simulationSnap;
protected virtual void Update();
```

The **Update** method provides the base lockstep mechanism. For every simulation turn **simulationSnap**, this function waits for the packets from all players to arrive and dispatches their actions to the **ExecuteAction** method before incrementing simulationSnap allowing the next turn to begin.

Simulation

```
protected abstract void ExecuteAction(IAction action);
protected abstract void OnSimulationStarted();
public void AddAction(IAction action);
```

Whenever the lockstep advances a turn of the simulation, **ExecuteAction** is called for every action of every player. This method must be overridden to implement the deterministic simulation of those actions. The **AddAction** method is used to push a new action to the queue of actions to be sent to other players. **OnSimulationStarted** event is raised whenever the simulation is initialized. It can be used for further initialization of the game (eg: spawning objects).

Sending data

```
public uint commandSnap;
private void SendSnap();
public void StartSimulation();
public void CreateServer();
public void ConnectToServer(HostData hostData);
```

Since Unity Networking does not implement a fully distributed peer-to-peer networking model, we abstract it by using RPCs that broadcast the messages to the other players. The **SendSnap** method is called repeatedly at regular interval and takes care of sending the actions for the current **commandSnap** (a counter for the last action sent). **CreateServer** and **ConnectToServer** are used to establish the underlying connection. And finally the **StartSimulation** method is invoked by the player acting as server to start the simulation on all players.

Lockstep settings

```
public class LockstepSettings : SingletonBehaviour<LockstepSettings> {
 public float snapSendInterval = 0.1f;
 public uint snapActionDelay = 2;
 public int portNumber = 1337;
 public int maxConnections = 8;
}
```

This singleton class defines the settings for the Lockstep mechanism:

- **snapSendInterval**: frequency of the network broadcast of the packets
- **snapActionDelay**: delay of execution of the actions
- **portNumber**: port number of the server
- **maxConnections**: maxium number of connections supported by the server

The actions are added to the packets that are added to the send queue of the LockstepPeer. In order to compensate for the time it takes for clients to communicate, a delay is added to when the action is actually executed in game*. This delay is defined by snapActionDelay and is measured in number of snaps. This value can be converted into time by considering the simulation rate of your game, and should be set according to the expected latency between the peers. For instance, if your game runs the simulation every 50 ms (20 FPS simulation), and the maximum expected latency between the peers is 150 ms, *snapActionDelay* should be set to 3. A higher number will add extra delay and eventually result in the game feeling less responsive, while a lower number may cause the game to temporarily stop on a client that is missing information from one of its peers.

Note that this is usually decoupled from the UI response to the user, which can right away give the illusion of a prompt execution (for example playing a "Yes, sir!" sound), even if the actual action will be executed later.

Packets and Actions

Packets

```
public interface IPacket : IEnumerable<IAction>{
  void AddAction(IAction action);
  int ActionCount { get; }
  string GUID { get; set; }
  uint Snap { get; set; }
}
```

The packets are defined by the **IPacket** interface, implemented by the **Packet** class. A packet contains a list of **IAction** that are defining the different messages sent to the other players for a snap. Actions can represent inputs given by the player (player selected n units) or events of the game (eg: unit x attacks y). Every packet contains the **GUID** of the player who emitted the action, and the simulation **Snap** that defines when the action is meant to be executed.

Actions

```
public interface IAction {
  int ObjectID { get; set; }
  int Action { get; set; }
  string GUID { get; set; }
}
```

Actions are defined by the **IAction** interface, implemented by the **AbstractAction** class. The user of this SE should then define its own Action by extending this class.

Packet Serialization

```
public interface IPacketSerializer {
  IPacket Deserialize(object msg);
  object Serialize(IPacket packet);
}
```

The **IPacketSerializer** interface defines a generic way to serialize the packets and their actions. It is currently implemented using the **BinaryFormatterPacketSerializer** (.net BinaryFormatter). You can easily reimplement this interface providing your own serialization method.

Additional notes

Deterministic simulation

If the game simulation is fully deterministic, the RTS-Lockstep guarantees a synchronized game state across all clients. This must be enforced by providing a deterministic AI, physics engine, path-finding and whatever component your game uses. This is harder than it might seem at the first place, as for instance floating point computations are in general not deterministic, because the IEEE 754 specification leaves the implementation free to compute differently the least significant bits. Thus the same computation might give different results on two processors, even from the same manufacturer.

In case of doubt, you might define a Checksum IAction that provides support for checking if the game state is the same on all clients. This is done by defining a number of game-state properties one wants to check (like the position of all units), and then computing a unique hash tag, which can be compared to the one of the other peers.

In case a mismatch is detected, it is up to the game developer to choose what strategy to adopt. The first problem is to establish an authority who can rule on the actual game state. This may not be possible if only two clients are present. When more peers are present, hopefully two or more will agree, simplifying the choice. The second problem is how to reconstruct the proper game-state in the de-synchronized clients. If the game-state is relatively small, it can be transferred from the other clients. If the game state is too large, a client could re-simulate locally the game state starting from the de-synchronization point. Both solutions may not be practical depending on your case, and often developers prefer to deal with the problem of making the simulation fully deterministic, rather than dealing with re-synchronization.

Relation with the Networked Virtual Character SE

The Networked Virtual Character SE is specialized to synchronize bones of 3-D characters between a client and a server, and is specific to the transmission of these information. On the contrary, the Game Synchronization SE implements a industry-standard way of synchronizing many game clients in a peer-to-peer fashion (lock-step mechanism) and is agnostic with respect to the types of data transmitted.